

PATENT ABSTRACTS OF JAPAN

(11)Publication number : 07-225703

(43)Date of publication of application : 22.08.1995

(51)Int.Cl.

G06F 11/28

G06F 9/06

G06F 11/30

G06F 12/14

(21)Application number : 07-039426

(71)Applicant : AT & T CORP

(22)Date of filing : 06.02.1995

(72)Inventor : GOODNOW II JAMES E
KOWALSKI THADDEUS J
ROWLAND JAMES R

(30)Priority

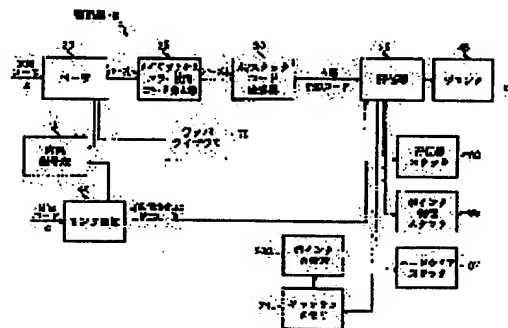
Priority number : 94 192239 Priority date : 04.02.1994 Priority country : US

(54) DETECTION OF ERROR IN SOFTWARE PROGRAM

(57)Abstract:

PURPOSE: To permit a software program to detect an error for de-referring to an invalid pointer.

CONSTITUTION: Pointer information containing the content of a pointer which is known last and the valid memory range of the pointer is stored in a cache memory 74 on the respective pointers. Whenever the software program detects that a new value is substituted for the pointer, pointer information is updated. Whenever the software program de-refers to the pointer, an execution time pointer inspection sub-routine is mounted. The sub-routine retrieves pointer information on the de-referred pointer, obtains the actual content of the pointer from a memory position where the pointer is stored and judges whether a memory space indicated by the content is within the valid memory range or not unless the actual content is equal to the content recorded in pointer information.



LEGAL STATUS

[Date of request for examination]

[Date of sending the examiner's decision of rejection]

[Kind of final disposal of application other than the
examiner's decision of rejection or application
converted registration]

[Date of final disposal for application]

[Patent number]

[Date of registration]

[Number of appeal against examiner's decision of

rejection]

[Date of requesting appeal against examiner's decision
of rejection]

[Date of extinction of right]

Copyright (C); 1998,2003 Japan Patent Office

(11)特許出願公開番号

(43)公開日 平成7年(1995)8月22日

【特許請求の範囲】

【請求項1】 テストされるソースコードファイルおよびそのソースコードファイルによって呼び出される1個以上のライブラリ関数からなり、メモリ位置を指示する1個以上のポインタを含み、ソフトウェア生成システムによって実行されるソフトウェアプログラムが、無効なポインタをデレファレンスするというエラーを検出する方法において、

前記ポインタのそれぞれについて、最後に既知となったポインタの内容と、ポインタの有効メモリ範囲とを含む

ポインタ情報を格納するステップと、
前記ソフトウェアプログラムによって前記ポインタのうちの1個以上のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびに前記ポインタ情報を更新するステップと、

前記ソフトウェアプログラムによってポインタのデレファレンスが行われることを前記ソフトウェア生成システムが検出するたびに実行時ポインタ検査サブルーチンを実装するステップとからなる方法であって、この実行時ポインタ検査サブルーチンが、

デレファレンスされたポインタに関するポインタ情報を検索する検索ステップと、

そのデレファレンスされたポインタが格納されているメモリ位置からそのポインタの実際の内容を取得するステップと、

その実際の内容が、前記ポインタ情報に記録されている当該デレファレンスされたポインタの最後に既知となった内容と等しい場合に限り、その実際の内容によって指示されているメモリ空間が、ポインタ情報から取得される有効メモリ範囲内にあるかどうかを判定するテストを行うテストステップとからなることを特徴とする、ソフトウェアプログラムのエラー検出方法。

【請求項2】 ポインタを作成することが既知であるライブラリ関数の実行後に、そのライブラリ関数によって作成されるポインタに対するポインタ情報を格納するステップを実行する実行後ラップ関数を実行するステップをさらに有することを特徴とする請求項1の方法。

【請求項3】 前記ソースコード内のポインタに代入するためのポインタ値を返値として返すことが既知であるライブラリ関数の実行後に、前記ソースコード内の当該ポインタへのその返値の代入を検出したときのポインタ情報を更新するステップの前に、前記ソフトウェア生成システムが後で使うためにその返されたポインタに関する更新されたポインタ情報をポインタ保存スタックに格納するステップを実行する実行後ラップ関数を実行するステップをさらに有することを特徴とする請求項1の方法。

【請求項4】 ポインタをデレファレンスすることが既知であるライブラリ関数を実行する前に、前記実行時ポインタ検査サブルーチンを開始する実行前ラップ関数を

実行するステップをさらに有することを特徴とする請求項1の方法。

【請求項5】 制限された範囲を有する1個以上の引数を受け取るライブラリ関数を実行する前に、その受け取った引数とその制限された範囲内にあるかどうかをテストする実行前ラップ関数を実行するステップをさらに有することを特徴とする請求項1の方法。

【請求項6】 前記ポインタ情報がポインタのステータス表示をさらに含み、

前記テストステップで、前記デレファレンスされたポインタの実際の内容が前記デレファレンスされたポインタの最後に既知となった内容と等しくない場合に前記ソフトウェア生成システムによって検出されなかったライブラリ関数の実行中の代入によって前記デレファレンスされたポインタの値が変更されたことを示すように、前記デレファレンスされたポインタに関するポインタ情報内のステータス表示を設定するステップをさらに有することを特徴とする請求項1の方法。

【請求項7】 前記ソフトウェアプログラムの実行後に前記ソフトウェア生成システムによって検出されなかったライブラリ関数の実行中の代入によって前記デレファレンスされたポインタの値が変更されたことを示しているステータスを有するポインタを識別するステップと、ライブラリ関数によるポインタの変更を検出する実行後ラップ関数が必要であることをプログラムに示すステップとをさらに有することを特徴とする請求項6の方法。

【請求項8】 前記ポインタ情報がポインタのアドレスをさらに含み、

前記検索ステップが、前記デレファレンスされたポインタに関するポインタ情報を識別するためにそのポインタ情報に含まれるポインタアドレスを検索するステップをさらに有することを特徴とする請求項1の方法。

【請求項9】 前記方法が、前記ソースコードに対応する構文木に対して読み出し時エラー検査プロセスを実行するステップをさらに有し、その読み出し時エラー検査プロセスが、

前記ソースコード内に現れるポインタ代入ごとに、ポインタテーブル更新コマンドと、ポインタに代入されるポインタ式を特徴づける特徴情報とを、前記構文木に挿入する読み出し時ポインタテーブル更新サブルーチンを実行するステップと、

前記ソースコードに現れるポインタのデレファレンスごとに、前記構文木にポインタ検査コマンドを挿入する読み出し時ポインタ検査サブルーチンを実行するステップとからなることを特徴とする請求項1の方法。

【請求項10】 前記構文木に対してデータフロー解析を実行するステップをさらに有することを特徴とする請求項9の方法。

【請求項11】 前記データフロー解析の結果を利用して、前記構文木から、重複するポインタ検査コマンドを

除去することを特徴とする請求項10の方法。

【請求項12】 前記データフロー解析の結果を利用して、不正なポインタのデレファレンスを実行時に検出することを特徴とする請求項10の方法。

【請求項13】 前記データフロー解析の結果を利用して、未初期化ポインタのデレファレンスを実行時に検出することを特徴とする請求項10の方法。

【請求項14】 前記構文木を内部擬似コードに変換するステップをさらに有することを特徴とする請求項9の方法。

【請求項15】 前記方法が、前記内部擬似コードに対して実行時エラー検査プロセスを実行するステップをさらに有し、この実行時エラー検査プロセスが、読み出し時にポインタ代入ごとに前記構文木に挿入されたポインタテーブル更新コマンドおよび特徴情報を評価したときに、前記ポインタ情報を格納するステップおよび更新するステップを実行する、実行時ポインタテーブル更新サブルーチンを実行するステップと、読み出し時にポインタのデレファレンスごとに前記構文木に挿入されたポインタ検査コマンドを評価したときに、前記実行時ポインタ検査サブルーチンを実行するステップとからなることを特徴とする請求項14の方法。

【請求項16】 テストされるソースコードファイルおよびそのソースコードファイルによって呼び出される1個以上のライブラリ関数からなり、メモリ位置を指示する1個以上のポインタを含み、ソフトウェア生成システムによって実行されるソフトウェアプログラムが、割当てメモリ領域を解放した後にその領域を指示するポインタをデレファレンスすることによりその領域にアクセスするというエラーを検出する方法において、

前記割当てメモリ領域を指示するポインタのチェインリストを含む、前記割当てメモリ領域に関する割当て情報を格納するステップと、

前記ソフトウェアプログラム内のポインタが前記割当てメモリ領域を指示している場合に、前記割当て情報への相互参照を含む、ポインタに関するポインタ情報を格納するステップと、

前記ソフトウェアプログラムによって前記ポインタのうちの1個以上のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびに前記ポインタ情報を更新するステップと、

前記チェインリストを変更するようなポインタ代入が検出されるごとに、前記チェインリストを更新するステップと、

前記割当てメモリ領域の解放を前記ソフトウェア生成システムが検出した場合に、対応する割当て情報からチェインリストを取得するステップと、

当該チェインリスト内にリストされた各ポインタに関するポインタ情報内に、そのポインタは解放したメモリ空間を指示しているという表示を記録するステップと、

ポインタがデレファレンスされるごとに対応するポインタ情報を評価して、そのポインタが解放されたメモリ空間を指示しているという表示があるにもかかわらずデレファレンスされているのかどうかを判定するステップとからなることを特徴とする、ソフトウェアのエラー検出方法。

【請求項17】 テストされるソースコードファイルおよびそのソースコードファイルによって呼び出される1個以上のライブラリ関数からなり、メモリ位置を指示する1個以上のポインタを含み、ソフトウェア生成システムによって実行されるソフトウェアプログラムが、メモリ領域の未初期化部分を指示するポインタをデレファレンスすることによりその未初期化部分を読み出すというエラーを検出する方法において、

メモリの各バイトに対する初期化ステータスを含む、前記メモリ領域に関する割当て情報を格納するステップと、

前記ソフトウェアプログラム内のポインタが前記メモリ領域を指示している場合に、前記割当て情報への相互参照を含む、ポインタに関するポインタ情報を格納するステップと、

前記ソフトウェアプログラムによって前記ポインタのうちの1個以上のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびに前記ポインタ情報を更新するステップと、

前記ソースコードファイルがポインタをデレファレンスすることによってメモリのバイトにアクセスしてそのバイトを初期化することを前記ソフトウェア生成システムが検出するたびに前記初期化ステータスを更新するステップと、

ライブラリ関数によって前記メモリ領域の1個以上のバイトが初期化されることを検出して、そのような初期化が検出された場合に前記初期化ステータスを更新する初期化ビットベクタ保守サブルーチンを実行するステップと、

ポインタがデレファレンスされて前記メモリ領域の1個以上のバイトが読み出されるたびに前記割当て情報内の初期化ステータスを評価して、読み出されるすべてのバイトが初期化されているかどうかを判定するステップとからなることを特徴とする、ソフトウェアのエラー検出方法。

【請求項18】 テストされるソースコードファイルおよびそのソースコードファイルによって呼び出される1個以上のライブラリ関数からなり、割当てメモリ領域を指示する可能性のある1個以上のポインタを含み、ソフトウェア生成システムによって実行されるソフトウェアプログラムが、割当てメモリをリークするというエラーを検出する方法において、

前記割当てメモリ領域を指示するポインタのチェインリストを含む、前記割当てメモリ領域に関する割当て情報

を格納するステップと、

前記ソフトウェアプログラム内のポインタが前記割当てメモリ領域を指示している場合に、前記割当て情報への相互参照を含む、ポインタに関するポインタ情報を格納するステップと、

前記ソフトウェアプログラムによって前記ポインタのうちの1個以上のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびに前記ポインタ情報を更新するステップと、

前記チェインリストを変更するようなポインタ代入が検出されるごとに、前記チェインリストを更新するステップと、

チェインリストが空である場合にメモリリークであると識別するステップとからなることを特徴とする、ソフトウェアのエラー検出方法。

【請求項19】 ソフトウェア生成システムによって実行されるソフトウェアプログラムによって呼び出される関数に渡されるポインタであって前記呼び出される関数内の形式引数に代入される実引数に関連し前記ソフトウェアプログラムおよび前記呼び出される関数内のポインタごとに前記ソフトウェア生成システムによってポインタ検査テーブルに保持されるポインタ情報を前記ソフトウェアプログラムから前記呼び出される関数に渡す方法において、

前記実引数に関連する渡されるポインタ情報の一時的格納用にメモリ位置を割り当てるステップと、

前記ポインタ検査テーブルから、前記実引数に関連するポインタ情報を取得するステップと、

取得したポインタ情報を、前記ソフトウェアプログラムによって前記呼び出される関数が呼び出されたときに前記一時的格納用メモリ位置に格納するステップと、

前記呼び出される関数の開始とともに、前記一時的格納用メモリ位置から前記ポインタ情報を取得するステップと、

前記呼び出される関数内の形式ポインタ引数に関連するポインタ検査テーブル内のエントリに、取得したポインタ情報を挿入するステップとからなることを特徴とする、ソフトウェアプログラムから呼び出される関数にポインタ情報を渡す方法。

【請求項20】 ソフトウェア生成システムによって実行されるソフトウェアプログラムによって呼び出された関数からそのソフトウェアプログラムに返されるポインタ値に関連し前記ソフトウェアプログラムおよび前記呼び出された関数内のポインタごとに前記ソフトウェア生成システムによってポインタ検査テーブルに保持されるポインタ情報を返す方法において、

前記呼び出された関数によって返されるポインタ情報に関連するポインタ情報の一時的格納用にメモリ位置を割り当てるステップと、

前記ポインタ検査テーブルから、前記呼び出された関数

から返されるポインタ値に関連するポインタ情報を取得するステップと、

取得したポインタ情報を、前記呼び出された関数から前記ソフトウェアプログラムに復帰したときに前記一時的格納用メモリ位置に格納するステップと、

前記呼び出された関数への復帰後に、前記一時的格納用メモリ位置から前記ポインタ情報を取得するステップと、

前記ソフトウェアプログラムの後続の実行中に、前記返されたポインタ値が代入されたポインタに関連するポインタ検査テーブル内のエントリに、取得したポインタ情報を挿入するステップとからなることを特徴とする、呼び出された関数からソフトウェアプログラムにポインタ情報を返す方法。

【請求項21】 メモリ位置を指示する1個以上のポインタを含むソフトウェアプログラム内の各ポインタに関するポインタ情報を保持するソフトウェア生成システムによって実行されるソフトウェアプログラム内の、それぞれポインタ式を有する複数の命令ストリームを含みそれらの命令ストリームのうちの1つのみが実行時にソフトウェア生成システムによって実行されその結果が実行時にデレファレンスされる複合式に対して、ポインタ検査を実行する方法において、

ポインタ情報の一時的格納用メモリ位置を割り当てるステップと、

前記ソフトウェア生成システムが、実行される命令ストリーム内のポインタに関するポインタ情報を取得し、取得したポインタ情報を前記一時的格納用メモリ位置に挿入するというコマンドを各命令ストリームに挿入するステップと、

実行時に前記複合式のポインタがデレファレンスされたときに前記一時的格納用メモリ位置からポインタ情報を取得するステップとからなることを特徴とするポインタ検査方法。

【請求項22】 メモリ位置を指示する1個以上のポインタを含み、各ポインタの有効メモリ範囲の情報を各ポインタに関するポインタ情報を保持するソフトウェア生成システムによって実行されるライブラリ関数が、無効なポインタをデレファレンスするというエラーを検出する方法において、

前記ライブラリ関数によってデレファレンスされる各ポインタに関するポインタ情報を取得し、デレファレンスされるポインタの実際の内容によって指示されるメモリ空間がポインタ情報から取得した有効メモリ範囲にあるかどうかを判定するラップ関数を前記ライブラリ関数と関係づけるステップと、

前記ライブラリ関数を実行する前に前記ラップ関数を実行するステップとからなることを特徴とする、ソフトウェアプログラムのエラー検出方法。

【請求項23】 メモリ位置を指示する1個以上のポイ

ンタを含みソフトウェア生成システムによって実行されるライブラリ関数の実行中に作成されるポインタに関するポインタ情報を更新する方法において、

前記ライブラリ関数によって作成されるポインタに関するポインタ情報を格納するラップ関数を前記ライブラリ関数と関係づけるステップと、

前記ライブラリ関数の実行後に前記ラップ関数を実行するステップとからなることを特徴とする、ポインタ情報の更新方法。

【請求項24】 ソフトウェア生成システムによって実行され、メモリ位置を指示する1個以上のポインタを含むライブラリ関数およびソースコードについて、ライブラリ関数からソースコードに返されソースコード内のポインタに代入されるポインタ値に関連しポインタごとにポインタ検査テーブルに保持されるポインタ情報を更新する方法において、

返されるポインタ値に関するポインタ情報をメモリ位置に格納するラップ関数を前記ライブラリ関数と関係づけるステップと、

関係づけられたライブラリ関数の実行後に前記ラップ関数を実行するステップと、

返されたポインタ値が前記ソースコード内のポインタに代入されたときに、前記メモリ位置からポインタ情報を取得して前記ポインタ検査テーブルに挿入するステップとからなることを特徴とするポインタ情報の更新方法。

【請求項25】 テストされるソースコードファイルおよびそのソースコードファイルによって呼び出される1個以上のライブラリ関数からなり、メモリ位置を指示する1個以上のポインタを含み、実行するソフトウェアプログラム内の各ポインタに関するポインタ情報を保持するソフトウェア生成システムによって実行されるソフトウェアプログラムに対して実行されるポインタ検査の数を最小にするためにソフトウェアプログラムをデータフロー解析する方法において、

前記ソフトウェアプログラム内の各ポインタに対するフローセットを設定するステップと、

前記ソフトウェアプログラムの各行に対して解析ルーチンを実行するステップと、

ポインタのデレファレンスを有するフローセットごとに、拡張オフセットを有する1回のポインタ検査を実行するステップとからなり、前記解析ルーチンが、解析しているソフトウェアプログラムの行内のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびに新しいフローセットを設定するステップと、

デレファレンスされるポインタに対する現在のフローセットに、そのデレファレンスされるポインタが解析しているソフトウェアプログラムの前記行内でデレファレンスされたことの表示によりマークするステップと、

前記ソフトウェアプログラムの各行に対して前記解析ル

ーチンを繰り返すステップとからなることを特徴とする、ソフトウェアプログラムをデータフロー解析する方法。

【請求項26】 メモリのバイトごとに初期化ステータスを保持するソフトウェア生成システムによって実行されるソフトウェアルーチンによるメモリのバイトの初期化を検出する方法において、

前記ソフトウェアルーチンがアクセスする可能性のあるすべての領域のバイトを識別するステップと、

前記領域の各バイトを既知の値でマークするステップと、

すべてのアクセス可能メモリに対する初期化ステータスを解析し、前記ソフトウェアルーチンが実行されるときに未初期化であるすべてのバイトを識別するステップと、

前記ソフトウェアルーチンが実行される前にすべての未初期化メモリの第1のチェックサムを実行するステップと、

前記ソフトウェアルーチンが実行された後にすべての未初期化メモリの第2のチェックサムを実行するステップと、

第1のチェックサムが第2のチェックサムに等しくない場合に前記ソフトウェアルーチンによるメモリの初期化があったと検出するステップとからなることを特徴とする、ソフトウェアルーチンによるメモリの初期化を検出する方法。

【請求項27】 メモリのバイトごとに初期化ステータスを保持するソフトウェア生成システムによって実行されるソフトウェアルーチンによるメモリのバイトの初期化を検出する方法において、

前記ソフトウェアルーチンがアクセスする可能性のあるすべての領域のバイトを識別するステップと、

すべてのアクセス可能メモリに対する初期化ステータスを解析し、前記ソフトウェアルーチンが実行されるときに未初期化であるすべてのバイトを識別するステップと、

アクセス可能なすべての未初期化メモリを保存するステップと、

前記ソフトウェアルーチンが実行される前にすべての未初期化メモリの第1のチェックサムを実行するステップと、

前記ソフトウェアルーチンが実行された後にすべての未初期化メモリの第2のチェックサムを実行するステップと、

第1のチェックサムが第2のチェックサムに等しくない場合に前記ソフトウェアルーチンによるメモリの初期化があったと検出するステップとからなることを特徴とする、ソフトウェアルーチンによるメモリの初期化を検出する方法。

【請求項28】 割当てメモリ領域を指示する1個以上

のポインタを含み、ソフトウェア生成システムによって実行されるソフトウェアプログラムが割当てメモリ領域を解放する方法において、

前記割当てメモリ領域を指示する各ポインタのステータス表示およびチェインリストを含む、前記割当てメモリ領域に関する割当て情報を格納するステップと、

前記ソフトウェアプログラム内のポインタが前記割当てメモリ領域を指示している場合に、ステータス表示および前記割当て情報への相互参照を含む、ポインタに関するポインタ情報を格納するステップと、

前記ソフトウェアプログラムによって前記ポインタのうちの1個以上のポインタに新しい値が代入されることを前記ソフトウェア生成システムが検出するたびごとに前記ポインタ情報を更新するステップと、

前記チェインリストを変更するようなポインタ代入が検出されるごとに、前記チェインリストを更新するステップと、

前記割当てメモリ領域の解放を前記ソフトウェア生成システムが検出した場合に、対応する割当て情報からチェインリストを取得するステップと、

当該チェインリスト内にリストされた各ポインタに関するポインタ情報のステータス表示内に、そのポインタは解放したメモリ空間を指示しているという表示を記録するステップとからなることを特徴とする、割当てメモリ領域を解放する方法。

【発明の詳細な説明】

【0001】

【産業上の利用分野】本発明は、ソフトウェアプログラムのテストおよびデバッグ処理システムに関し、詳しくは「翻訳された原始（ソース）コード」と「コンパイルされた目的（オブジェクト）コード」との両方におけるメモリアクセス誤り（エラー）を検出するための方法および装置に関する。

【0002】

【従来の技術】今日のコンピュータプログラミングの世界においては、多くの高水準のプログラミング言語が利用できる。しかし、その中でC、C++、およびPascalのようなプログラミング言語は、それらの持つ柔軟性およびパワーによってプログラマの間で非常に人気のある言語となっている。

【0003】これらのプログラム言語には、ソフトウェアにおいてプログラムが実現できることについての制約がもしあったとしても非常に少ないので、根底にあるアセンブリプログラムが行うタスクであれば実際上どのようなタスクでもプログラムに行わせることが可能である。

【0004】顕著なパワーおよび柔軟性の得られるこれらのプログラミング言語の特徴の1つは、ポインタによって制約なしにメモリにアクセスできるという能力である。しかしポインタを制約なしに使用できることから、

従来のデバッグ処理手法では検出および訂正がしばしば困難であるようなプログラムバグが生じる。

【0005】種々のメモリアクセスエラーの検出用に、いくつものソフトウェアテストおよびデバッグ処理ツールが開発されている。例えば、米国カリフォルニア州サニーベイルのビュアソフトウェア・インコーポレーテッドから商業的に入手可能で、米国特許第5,193,180号に開示されている、商品名「ピュリファイ(Purify)」ソフトウェアテストツールは、メモリアクセスエラーおよびメモリ漏洩（リーク）を検出するためのシステムを提供する。この「ピュリファイ」システムは、メモリの各バイトについてメモリ割付および初期化の状態を監視（モニタ）する。

【0006】これに加えて、「ピュリファイ」システムは、メモリ要素配列（メモリ配列）境界違反および同様のメモリアクセスエラーを検出するために、割付を受けたメモリの各ブロックの前後に8バイトのバッファゾーンを設けている。バッファゾーン内の各バイトの状態は、割付も初期化もされていない状態（非割付・非初期化状態）にセットされる。

【0007】メモリにアクセスする各命令について、「ピュリファイ」システムは、割り付けられていないメモリへの書き込み、および初期化されていないまたは割り付けられていないメモリからの読み出しをプログラムが行っていないことを確実にするための試験を行う。

【0008】「ピュリファイ」システムを用いることによって、多くのメモリアクセスエラーを検出するための有効な基礎は得られるが、このシステムは、割付されたメモリの第1のブロックに関連するポインタが、割付され初期化されたメモリの第2のブロックに不適切にアクセスする場合に生じる共通プログラムエラーの検出はしない。

【0009】すなわち、「ピュリファイ」システムは、ポインタによって指示されたメモリが割付されていることおよび初期化されていることだけを検証し、ポインタによって指示されたメモリがそのポインタについて設定されている適切な境界範囲内にあることの検証は行わない。

【0010】この制限を克服しようとする試みから、別のソフトウェアテストおよびデバッグ処理ツールが開発された。例えば、ステッフェン(Joseph L. Steffen)の論文「ポータブルCコンパイラへの実行時（ランタイム）検査の追加」(Software-Practice and Experience 誌、第22(4)巻、1992年4月、305-316ページ)に記述されている、コンパイラを基盤とするメモリアクセス検出システムは、各ポインタについて3語を利用し、それによってポインタがそのポインタの有効範囲についての情報を得られるようにしたものである。

【0011】すなわち、ポインタがメモリにアクセスする都度、そのポインタによって指示されたメモリがそれ

それぞれのポインタについての適切な境界範囲内にあることを確実にするような検査が行われる。

【0012】しかし、プログラマの多くは、自分のソフトウェアのテストおよびデバッグ処理を、コンパイラ環境よりも翻訳器（インタプリタ）環境で行うことを好む。その理由は、インタプリタを基盤とするデバッグ処理の方が一般に、程度の高い追跡およびその他の診断手法が得られることから、デバッグ処理中により柔軟性がありまたより大きな支援が得られるためである。

【0013】たぶん、より普通なのは、いくつかのファイルが「翻訳された原始コード」から構成され他のファイルが「コンパイルされた目的コード」から構成されるような、部分的翻訳である。

【0014】しかし、部分的翻訳の環境で作動するソフトウェアデバッグ処理ツール（例えば、センタラインソフトウェア・インコーポレーテッドから商業的に入手可能な、前に「セバ・C」システムとして知られた「センタラインコードセンタ」システム）は一般に、「コンパイルされた目的コード」のエラー検査に関しては、上に述べた「ビュリファイ」システムと同じ制限がある。

【0015】詳しくは、これらの部分的翻訳デバッグ処理ツールは一般に、「コンパイルされた目的コード」内でポインタによって指示されたメモリが割付され且つ初期化されていることだけを検証し、ポインタによって指示されたメモリがそのポインタについて設定されている適切な境界範囲内にあることの検証は行わない。

【0016】

【発明が解決しようとする課題】従来の技術についての上記の欠点から明らかなように、「翻訳された原始コード」と「コンパイルされた目的コード」との両方を実行しながらエラー検出タスクを行うことのできるソフトウェアテストおよびデバッグ処理ツールの必要性が存在する。さらに、与えられたポインタによって指示されたメモリがそれぞれのポインタについての適切な境界範囲内にあることを確実にするソフトウェアテストおよびデバッグ処理ツールの必要性が存在する。

【0017】その上、読み出し時またはパース時に導出される情報を利用することにより、実行時に行われる重複または重合するポインタ検査の数を減少させるような、より効率的なソフトウェアテストおよびデバッグ処理ツールの必要性が存在する。

【0018】

【課題を解決するための手段】概略的には、本発明の形態によれば、ソフトウェアプログラムにおけるいくつかのメモリアクセスエラー、を検出するためのソフトウェアテストおよびデバッグ処理システムとしての処理ツールが設けられる。これらのメモリアクセスエラーは例えば、配列次元違反、無効ポインタの「関連付けの解除（関連解除）（デレファレンス）」、解放されたメモリへのアクセス、初期化されていないメモリからの読み出

し、およびメモリ漏洩の自動検出である。これに加えて、本システムは、検出されたエラーの特定および訂正を行うための診断情報を有する。

【0019】本発明の別の態様によれば、ソフトウェアプログラムにおける各ポインタについてのポインタ情報を記録するためのシステムが設けられる。記録されたポインタ情報は、例えば連関するポインタのアドレスおよび内容、ならびに連関するポインタが有効に指示できるメモリ範囲を含むのが好ましい。

【0020】これに加えて、割付されたメモリの各領域についての割付情報を記録するためのシステムが設けられる。記録されたメモリ割付情報は、例えば割付されたメモリの連関領域を現に指示する全ポインタのチェインリスト、および割付されたメモリの連関領域の各バイトの初期化状態を維持する初期化ビットベクトルを含むのが好ましい。

【0021】読み出し時エラー検査プロセスが、翻訳された原始コードに連関する構文木（パース木）の各ラインを解析する。翻訳された原始コードに現れる各ポインタ割当またはポインタ関連解除の各々に応答して、「読み出し時エラー検査プロセス」が、パース木内にエラー検査コマンドおよび追加情報を適切に挿入する。

【0022】「実行時エラー検査プロセス」が、ポインタ割当に応答してパース木内に挿入された各エラー検査コマンドについての適切な、記録されたポインタ情報の更新を開始する。同様に、ポインタ関連解除に応答してパース木内に挿入された各エラー検査コマンドについて、「実行時エラー検査プロセス」が、関連解除（デレファレンス）されたポインタについて記録されているポインタ情報のポインタ検査を開始する。

【0023】実行時ポインタ検査は、関連解除されたポインタについての記録されているポインタ情報を検討評価（または単に、評価）し、関連解除されたポインタがその有効メモリスペース外を指示しているかどうかを検出する。

【0024】本発明のさらに別の特徴によれば、「コンパイルされた目的コード関数」（または簡単に、「コンパイルされたコード関数」）に対するエラー検査プロセスが、エラー検査を要する「コンパイルされたコード関数」に連関する翻訳された「ラップ関数」を用いて行われる。必要なエラー検査プロセスを実現するために、ラップ関数は、必要に応じて、連関する「コンパイルされたコード関数」の前および/または後に実行される。

【0025】関連解除されたポインタの各々についてポインタ検査を開始するために、「コンパイルされた目的コード」の実行中ポインタを関連解除するものとして知られる「コンパイルされたコード関数」の各々に「実行前ラップ関数」を連関させるのが好ましい。「実行前ラップ関数」は、「コンパイルされたコード関数」へ送られた引数（アーギュメント）についての追加検査も行

う。

【0026】「コンパイルされたコード」の実行中ポインタの生成を行うものとして知られる「コンパイルされたコード関数」の各々に「実行後ラップ関数」を連関させるのが好ましく、同様に、ポインタ値を返すものとして知られる「コンパイルされた目的コード関数」の各々に「実行後ラップ関数」を連関させるのが好ましい。このポインタ値は次に、呼び出し関数によって返されるとすぐに別のポインタに割り当てられる。

【0027】もし連関させた「コンパイルされたコード関数」がポインタを生成する場合、
10 「実行後ラップ関数」が、生成されたポインタについてのポインタ情報を適切なメモリ位置に記録することが好ましい。もし連関させた「コンパイルされたコード関数」がポインタ値を呼び出し関数に返す場合、呼び出し関数の実行中に次に行われる検索取り出しのために、「実行後ラップ関数」が、必要なポインタ情報をポインタ保存スタックに位置させることが好ましい。

【0028】本発明のさらに別の特徴によれば、割付を解除されたメモリの領域がアクセスされる場合を検出する
20 ための方法が提供される。割付されたメモリの領域の割付の解除後すぐに、そのメモリ領域に連関するメモリ割付情報に含まれるチェインリストがアクセスされて、割付されたメモリの領域を現に指示する全ポインタのリストが得られる。その後、ポインタが、割付を解除されたメモリの領域を今指示している旨の表示が、チェインリスト中に列記された各ポインタに連関するポインタ情報に記録される。

【0029】したがって、もし、割付を解除されたメモリスペースをポインタが指示している旨の表示がありな
30 ながら、そのポインタが関連解除される場合に、エラーが検出される。

【0030】本発明の別の特徴によれば、初期化されていないメモリの領域からの読み出しが行われる場合を検出する方法が提供される。割付されたメモリの領域の1個以上のバイトが「翻訳された原始コード」によって初期化される都度、メモリのその連関領域についてのメモリ割付情報における初期化ビットベクトルが、その対応する数のメモリバイトが初期化されたことを表示するように更新される。

【0031】メモリを初期化する「コンパイルされたコード関数」が実行される都度、初期化ビットベクトル維持サブルーチンが行われる。初期化ビットベクトル維持サブルーチンは、「コンパイルされたコード関数」によってメモリの1バイト以上の初期化を検出し、その対応する数のメモリバイトが初期化されたことを表示するように初期化ビットベクトルを更新する。

【0032】メモリの領域からの読み出しを行うようにポインタが関連解除される都度、初期化ビットベクトルが評価されて、読み出しが行われているメモリ領域が初
50

期化された状態にあるかどうかが定められる。

【0033】本発明のさらに別の特徴によれば、メモリ漏洩が自動的に検出可能となる。割付されたメモリの各領域を現に指示しているポインタのリストがポインタの割当によって修正される都度、割付されたメモリのその領域に連関するメモリ割付情報に含まれるチェインリストが更新される。もしチェインリストが空の場合、メモリ漏洩が識別される。

【0034】本発明のなお別の特徴によれば、データ流れ解析を利用して、読み出し時パーズ木についての重複すなわち重合するポインタ検査を削除することによって、実行時において行う必要のあるポインタ検査の数を最小化する処理が行われる。これに加えて、データ流れ解析により、空のまたは初期化されていないポインタの関連解除を読み出し時において検出することが可能となる。

【0035】

【実施例】本発明に基づくメモリアクセスエラー検出システムを図1に示す。ここに開示するメモリアクセスエラー検出システムは、ソフトウェア生成システムを基盤とするソフトウェアテストおよびデバッグ処理ツールである。この場合のソフトウェア生成システムは、翻訳器（インタプリタ）、またはコンパイラ、あるいはその他、原始コードを実行可能なフォーマットに変換する能力を有する類似のシステムとして実施される。

【0036】本メモリアクセスエラー検出システムは、種々のメモリアクセスエラーを検出するために、検査対象ソフトウェアプログラムを解析する。これらのメモリアクセスエラーは例えば、配列次元違反、無効ポインタの関連解除、解放されたメモリへのアクセス、初期化されていないメモリからの読み出し、およびメモリ漏洩の自動検出である。加えて、本システムは、検出されたエラーの特定および訂正を行うための診断情報を有する。

【0037】本発明を以下5つの主要項目に分けて説明する。第1に、導入部において、まず図1の概略ブロック図を参照して、本発明のメモリアクセスエラー検出システムについて述べる。またこの導入部では、概略的には図1にまた詳しくは図2に示すようなポインタ検査表200についての詳細説明も行う。図2に示すポインタ検査表200は、図3～図6を参照して説明するような
40 連関事例におけるデータからなる。

【0038】第2に、図1に示すメモリアクセスエラー検出コード挿入器25によって実行されるような、本発明の「読み出し時エラー検査プロセス」を、〔読み出し時エラー検査動作〕と題する項目において、図7～図11を参照して説明する。本項目においてはその終りに、図23および図24を参照して、実行時において行う必要のあるポインタ検査の数を最小化することとある読み出し時エラーを検出できるようにすることとを目的として読み出し時において行われるデータ流れ解析、につい

て述べる。

【0039】第3に、「コンパイルされた目的コード関数」(または簡単に、「コンパイルされたコード関数」)についてのエラー検査を実現するための好ましい実施例を、[コンパイルされた目的コードのエラー検査]と題する項目において説明する。本項目においては、連関する「コンパイルされたコード関数」についてのエラー検査を行うために必要に応じて実行される「翻訳されたラッパ関数」に関して述べる。コンパイルされたメモリ割付解除関数の各々に連関する好ましい「実行前ラッパ関数」についても図12を参照して説明する。

【0040】第4に、図1に示す評価器35によって実行されるような、本発明の「実行時エラー検査プロセス」を、[実行時エラー検査動作]と題する項目において、図13～図22を参照して説明する。

【0041】最後に、メモリ漏洩を検査するための方法、および検出されていないポインタ修正のソースを「コンパイルされたコード関数」によって特定するための方法を、[メモリ漏洩の特定および検出されていないポインタ修正のソースの特定]と題する項目において説明する。

【0042】図1に示すように、メモリアクセスエラー検出システムは、商業的にAT&T社から入手可能な、Cプログラミング言語用のCIN翻訳器をエラー検査関数を得られるように改造した翻訳器のような、翻訳器(インタプリタ)15を用いるのが好ましい。下でさらに述べるように翻訳器15は「翻訳された原始コード」および「コンパイルされた目的コード」の両方を実行しながらエラー検出タスクを行う。

【0043】翻訳器15によって受信される原始コードを、読み出し時においてパーザ(構文解析器)20が解析し、原始コードを周知のパーザ木(構文木)に変換する。これに加えて、原始コードがプログラムメモリに読み込まれると、内部記号表75がパーザ20によって形成される。

【0044】内部記号表75には、原始コードに定義される記号ラベルの各々についてのエントリが既知の方法で含まれる。内部記号表75の各エントリからその連関する記号ラベルが判る。またこれらの各エントリは、それぞれの記号ラベルに割付されたメモリ位置のアドレスを含む。

【0045】メモリアクセスエラー検出コード挿入器25が設けられ、パーザ20によって生成されたパーザ木上で、図7～図11を参照して下にさらに説明するように読み出し時エラー検査プロセスを行う。下にさらに述べるように、メモリアクセスエラー検出コード挿入器25が、原始コードから導出されたパーザ木を解析し、適切な場合にはエラー検査コマンドおよび追加情報をパーザ木の中に挿入する。この追加情報は実行時において評価される。

【0046】それから、メモリアクセスエラー検出コード挿入器25によって生成された修正パーザ木(挿入されたエラー検査コマンドおよび追加情報から構成される)が、「木からスタックへコード変換する(木/スタックコード変換器)30」に入力される。木/スタックコード変換器30は、機械語内に本来ある内部疑似コードを生成する。

【0047】上に述べたように、翻訳器15は「翻訳された原始コード」および「コンパイルされた目的コード」の両方を実行しながらエラー検出タスクを行うことが好ましい。そのため、コンパイルされた目的コードを再配置するリンク装置40が設けられる。目的コードが読み出し時においてリンク装置40によってプログラムメモリにロードされると、内部記号表75が、目的コードにおいて定義された各記号ラベルについてのエントリを含むように、上に述べた方法で更新される。

【0048】実行時において、評価器35が、インタプリタスタック60を利用して、木/スタックコード変換器30によって生成された内部疑似コードと、翻訳された内部疑似コードから呼び出されたコンパイルされた目的コードとを、さらに下で述べる方法で実行する。評価器35の出力をモニタするために、プリンタ45が設けられる。このプリンタ45は、原始コードフォーマットで評価器35によって実行される内部疑似コードプログラムを列記する。

【0049】本発明の一特徴によれば、評価器35が、読み出し時中および実行時状況において内部疑似コード内に設定されたコマンドおよび情報に基づいて、図2を参照してさらに下で述べるように、ポインタ検査表200を生成する。ポインタ検査表200は、それぞれのポインタの使用および修正をモニタするために利用される各ポインタについての情報を記録する。

【0050】これに加えて、ポインタ情報を評価器35が将来使用することが予想されるのでそのために、ポインタ情報が、キャッシュメモリ74、すなわち非常に高速のメモリ区域に評価器35によって一時的に記憶される。本発明の一実施例においては、キャッシュメモリ74が最も最近に参照された2個のポインタのアドレスを記憶する。これらの値が、下に述べるように実行時における動作中に評価器35によって次に必要とされる場合、このような方法で、これらの値が評価器35へ直接に転送されるので、動作速度が増加する。

【0051】もし望むポインタ情報がキャッシュメモリ74内に見出されない場合、好ましくは、バケットが最も最近のポインタ更新によって分類されるような「バケット・ハッシュ・ルックアップ」アルゴリズムを用いて、ポインタ検査表200がアクセスされる。

【0052】例えば、評価器35によって実行されている関数の引数を記憶するために、周知のハードウェアスタック67が評価器35によってアクセスされる。これ

に加えて、評価器35が、実行時エラー検査動作の好ましい実施例を実現化しながら、ポインタ保存スタック65にアクセスする。これについてはさらに下に述べる。

【0053】さらに下の「コンパイルされた目的コードのエラー検査」と題する項目で述べるが、コンパイルされたコード関数に連関する「翻訳された実行前および実行後ラップ関数」を記憶するために、ラップ・ライブラリ70を設けることが好ましい。「翻訳された実行前および実行後ラップ関数」によって、連関するコンパイルされたコード関数について、必要に応じてエラー検査プロセスを行うことが可能となる。

【0054】さらに下で「実行時エラー検査動作」と題する項目において述べるように、ポインタ検査表200の適切なエントリが、ポインタが新しい値を割り当てられる都度、評価器35により実行時において更新される。この方法で、ポインタ検査表200が、各ポインタについての現ポインタ情報を記録する。

【0055】しかし、周知のように、従来の翻訳器は通常、コンパイルされた目的コードによってポインタ値に加えられた修正については気付かない。したがって、本発明の一態様により、ポインタ検査表200内の適切なエントリを修正ポインタ情報で更新するために、ポインタ値を修正するものとして知られる連関するコンパイルされたコード関数の実行後に、下の「コンパイルされた目的コードのエラー検査」と題する項目において説明する「実行後ラップ関数」を実行することが好ましい。

【0056】これに加えて、新しいポインタ値が次に用いられるときにそのポインタ値についての前に検出されなかった修正を検出してフラグを付けるためのステップ1124およびステップ1128（図18）における追補メカニズムが、下に図18および図19を参照して説明する実行時ポインタ検査サブルーチンによって得られる。

【0057】図2に示すポインタ検査表200のようなポインタ検査表は、異なるポインタに各々が連関する横列220、222、224のような、複数の横列を有する。ポインタ検査表200の各横列は、連関するポインタについての情報を記憶するための複数のエントリを有する。

【0058】エントリ230は、連関するポインタが記憶されているメモリ内のアドレスを記憶する。同様に、エントリ240は、連関するポインタの内容を記憶する。エントリ250aは、連関するポインタの有効下側メモリ境界を記録し、他方、エントリ250bは、有効上側メモリ境界を記録を記録する。状態エントリ260は、連関するポインタについての、さらに下に述べるいくつもの予め定義されたコードのうちの1つを記録する。

【0059】最後に、ファイル／ライン番号エントリ270は、ポインタの最後の修正に連関するファイル名お

よびライン番号を指示するポインタを記録する。この最後の修正データを利用してエラー検出時に診断情報が得られる。

【0060】説明の便宜上、図3に示す原始コードファイル例「テストファイル」のライン10からライン210までの実行後すぐに宣言され初期化されたポインタに連関するポインタ情報から構成されるものとする。なお、原始コードファイル「テストファイル」は、各整数変数について2バイトおよび各文字変数について1バイトを割付するマシン上で作動する。

【0061】図3に示す原始コードファイル「テストファイル」のライン10からライン40までの実行中、整数番号「number」および文字番号「name」を有する「part」型の構造体「widget」が宣言される。ポインタ「ptr_part」がライン50において宣言される。

【0062】ライン60の実行中、ポインタ「ptr_part」は、構造体「widget」のアドレスを指示するように割当を受ける。ライン60におけるこのポインタ指示割当によって、ポインタ「ptr_part」についてのポインタ情報がポインタ検査表200内に入れられる。ライン10からライン60までの実行の結果として得られるメモリ割付を図4に示す。

【0063】図4から判るように、ポインタ「ptr_part」にはアドレス「2048」がメモリ割付されており、ポインタ「ptr_part」についてのポインタ検査表200のポインタエントリ230のポインタアドレス欄（図2）に入れられる。ポインタ「ptr_part」は、構造体「widget」のアドレス、すなわちアドレス「1000」を指示するように割り当てられており、ポインタ「ptr_part」についてのポインタエントリ240のポインタ内容欄（図2）に入れられる。

【0064】ポインタ「ptr_part」は、構造体「widget」内のどの位置、すなわち範囲1000から1011までの領域内のどのアドレスを指示する割当も有効であるので、この情報は有効メモリ境界エントリ250a、250b内に入れられる。

【0065】ポインタ検査表200内の状態エントリ260は、ポインタ「ptr_part」が「境界あり」（BOUNDED）とマーク付けした領域を指示することを表すように更新されることが好ましく、下でさらに述べるような方法で更新されることが好ましい。ファイル／ライン番号エントリ270は、ポインタ「ptr_part」がライン60において原始コードファイル「テストファイル」によって前回更新されたことを表すように更新されることが好ましい。

【0066】図3に示す原始コードファイル「テストファイル」のライン70およびライン80の実行中、2個の変数、すなわち整数変数「testint」および文

字変数「testchar」がそれぞれ宣言される。ポインタ「ptr_testint」がライン90において宣言され、整数変数「testint」を指示するように割当を受ける。ポインタ「ptr_testchar」がライン100において宣言され、文字変数「testchar」を指示するように割当を受ける。

【0067】ライン90およびライン100におけるポインタ割当によって、ポインタ「ptr_testint」およびポインタ「ptr_testchar」についてのポインタ情報がポインタ検査表200内に入れられる。ライン70からライン100までの実行の結果として得られるメモリ割付を図5に示す。

【0068】図5から判るように、ポインタ「ptr_testint」にはアドレス「2086」がメモリ割付されており、ポインタ「ptr_testchar」にはアドレス「2090」がメモリ割付されている。この情報は、各ポインタ、すなわちポインタ「ptr_testint」およびポインタ「ptr_testchar」、についてのポインタ検査表200のポインタエントリ230のポインタアドレス欄（図2）に入れられる。

【0069】ポインタ「ptr_testint」は、整数変数「testint」のアドレス、すなわちアドレス「1500」を指示するように割り当てられており、ポインタ「ptr_testchar」は、文字変数「testchar」のアドレス、すなわちアドレス「1502」を指示するように割り当てられている。この情報は、各ポインタ、すなわちポインタ「ptr_testint」およびポインタ「ptr_testchar」、についてのポインタ検査表200のポインタエントリ240のポインタ内容欄（図2）に入れられる。

【0070】ポインタ「ptr_testint」は、2バイトの整数変数「testint」だけ、すなわち欄1500から1501までの領域内だけを指示する指示するので、この情報は、ポインタ「ptr_testint」についての有効メモリ境界エントリ250a、250b内に入れられる。

【0071】同様に、ポインタ「ptr_testchar」は、1バイトの文字変数「testchar」だけ、すなわち1バイトアドレス1502だけを指示する。したがって、この情報は、ポインタ「ptr_testchar」についてのポインタ検査表200内の有効メモリ境界エントリ250a、250b内に入れられる。

【0072】ポインタ「ptr_testint」およびポインタ「ptr_testchar」についてのポインタ検査表200内の状態エントリ260は、それぞれのポインタが「境界あり」の領域を指示することを表すように更新されることが好ましく、下でさらに述べるような方法で更新されることが好ましい。ファイル/ラ

イン番号エントリ270は、これらのポインタがライン90およびライン100において原始コード「テストファイル」によって前回更新されたことを表すようにそれぞれ更新されることが好ましい。

【0073】コンパイルされたライブラリ関数「malloc」を用いて35バイトのメモリの割付を行い、割付を受けたメモリの始点アドレスの値を、宣言されたポインタ「ptr_malloc」に返す、原始コードファイル「テストファイル」のライン200からライン210までの実行と、結果として得られるメモリ割付（図6に示す）とについて、下の「コンパイルされた目的コードのエラー検査」と題する項目において説明する。

【0074】図3に示す原始コードファイル「テストファイル」のライン300の実行中、ポインタ「ptr_testchar」（前に述べたようにライン100において前に値を割り当てられている）が、構造体「widget」の構成部分（メンバ）を指示するように指示について再割当される。したがって、ポインタ「ptr_testchar」についてのポインタ検査表200における情報については、指示割当に関連する新しいポインタ情報に更新する必要がある。

【0075】すなわち、ライン300の実行に続いて、ポインタ「ptr_testchar」についての「ポインタの内容」エントリ240を、ポインタが今、構造体構成部分（構造体メンバ）「widget.name」、つまり「1002」を指示することを表すように更新する必要がある。

【0076】なお、もしポインタ構成部分（ポインタメンバ）を含む第1の構造体が第2の構造体にコピーされる場合、第1の構造体におけるポインタメンバについてのポインタ検査表200内に記録されているポインタ情報を、第2の構造体におけるポインタメンバについてのエントリ内に適切に入れる必要がある。

【0077】本発明の一特徴によれば、ポインタ値の有効性を確実にするために、上に述べたポインタ検査表200が、ポインタが関連解除される都度、すなわちそのポインタによって指示されるメモリがアクセスされる都度、利用される。例えば、図3に示す原始コードファイル「テストファイル」のライン310の実行後すぐに、ポインタ「ptr_testint」によって指示されるアドレスに定数12024が書き込まれるようにポインタ「ptr_testint」が関連解除される。

【0078】下でさらに述べるように、ポインタ「ptr_testint」によって指示されるメモリ位置がアクセスされる直前に、ポインタ「ptr_testint」が有効なメモリ位置、すなわちポインタ「ptr_testint」についての有効なメモリ境界エントリ250a、250bに記録されているような有効なメモリ境界1500から1501までの範囲内、を指示していることを確実にするために、ポインタ内容の有効性

を検査する。メモリアクセスエラー検出システムが、

【0079】本発明の好ましい一実施例において、静的ポインタについての情報を記録する表と自動ポインタについての情報を記録する表との2個の別個のポインタ検査表200が維持される。自動ポインタは、自動ポインタが定義される特定関数の範囲内でだけ既知であるので、連関するポインタ情報を別個の表に記録する方が、より効率的である。したがって、自動ポインタ検査表200は、各関数の実行後、再初期化される。

【0080】ポインタ検査表200には、2種類のポインタ、すなわち静的データを指示するように初期化される静的ポインタと主関数へ送られる「argv」ポインタ引数とについてのポインタ情報をプリロードする（先に入れる）ことが好ましい。これらの原始コードファイルラインの実行は通常、下に述べるポインタ検査表更新のメカニズムに関連しないので、これらの情報については、内部疑似コードの実行に先立ってポインタ検査表200内に自動的にプリロードすることが好ましい。

【0081】「読み出し時エラー検査動作」メモリアクセスエラー検出コード挿入器25がパズ木の各ラインについて読み出しを行って、図7の流れ図に示す「読み出し時エラー検査プロセス」を起動実現させる。「読み出し時エラー検査プロセス」は、ある種のエラー検査コマンドと追加情報とをパズ木に挿入するある種の読み出し時エラー検査タスクを行うために、パズ木の各ライン、またはノード、を評価する。

【0082】この、メモリアクセスエラー検出コード挿入器25によって実現される「読み出し時エラー検査プロセス」は本来、図13から図22までを参照して下に説明するような、実行時において行われる実際のエラー検出プロセス、を起動し実行させる役をする前段階タスクを行う。

【0083】「読み出し時エラー検査プロセス」は、図7に示すように、もし評価中のコードラインについて実行すべきエラー検査サブルーチンがあれば、そのうちのどのエラー検査サブルーチンを実行すべきかを定める。概略的には、プロセスについてのプログラム実行手順（以下簡単に、手順）は、パズ木の各ラインが読み出されるステップ300において「読み出し時エラー検査プロセス」に入り、その後、もしステップ310の条件が満足される場合、「読み出し時配列次元検査サブルーチン」（図8を参照して下に述べる）が実行される。

【0084】同様に、もしステップ320の条件が満足される場合、「読み出し時ポインタ検査表更新サブルーチン」（図9および図10を参照して下に述べる）が実行される。またもしステップ330の条件が満足される場合、「読み出し時ポインタ検査サブルーチン」（図11を参照して下に述べる）が実行される。最後に、もしステップ335の条件が満足される場合、ステップ337の間に、適切なフラグ（下に述べる）がセットされ

る。

【0085】これに加えて、メモリアクセスエラー検出コード挿入器25がパズ木のラインのすべてを読み出して解析したことがステップ340において検出されると、「読み出し時エラー検査プロセス」は、ステップ350およびステップ360においてデータ流れ解析（下でさらに説明する）を行う。これによって、実行時において行う必要のあるポインタ検査の数が削減され、読み出し時エラーの検出が可能となる。

【0086】もし「読み出し時エラー検査プロセス」によって評価中の原始コードファイルのラインが、宣言された配列に対する参照を含むことがステップ310（図7）において検出された場合、手順はステップ410（図8）において図8に示す「読み出し時配列次元検査サブルーチン」に入り、同サブルーチンが開始される。

【0087】「読み出し時配列次元検査サブルーチン」は、不正な添字（指標）、すなわち負の添字または宣言された配列サイズを超過する添字が配列を参照するために用いられているかどうかを判定する。多くのプログラミング言語において、多次元配列のうちの1つの次元の範囲を超過しながらなお配列の全範囲内に収まることは可能なので、各次元についてその適切な境界が維持されているかどうかの検査が行われる。

【0088】なお、最大有効配列添字は一般に、その次元についての宣言された配列サイズよりも小さい。その理由は、多くのプログラミング言語について、配列の最初の有効添字が「0」であって「1」ではないからである。

【0089】図8の「読み出し時配列次元検査サブルーチン（または簡単に、配列次元検査サブルーチン）」は、ステップ410において、内部記号表75から配列の各次元についての、宣言されたサイズの最大値（最大宣言サイズ）を取得する。もし添字参照が可変の場合、添字参照の値は読み出し時においては判らず、実行時において参照が評価されるまでは評価できない。このため、配列のある次元についての添字参照が定数かまたは変数かを定める検査がステップ420においてまず行われる。

【0090】もしステップ420において、添字参照が定数であると判断された場合、ステップ430において、コードのラインに現れる定数値がこの次元についての最大値と比較される。もしステップ430において、定数参照がこの次元についての最大宣言値を超える値かまたは負の値である場合、ステップ435においてエラーメッセージが生成され、ステップ440において、手順はプロセスから出て、プロセスが終了する。

【0091】もしステップ430において、定数参照が有効であると判断された場合、手順はステップ450に進みここで、この配列に検査すべき他の次元があるかどうか、すなわちこの配列が多次元かどうか判定され

る。

【0092】もしステップ450において、検査すべき他の次元があると判断された場合、手順はステップ420に戻り、上に述べた方法で他の残りの次元についての検査が行われる。もしステップ450において、検査すべき他の次元が残っていないと判断された場合、手順はステップ320（図7）において「読み出し時エラー検査プロセス」に戻る。

【0093】もしステップ420（図8）において、添字参照が可変であると判断された場合、ステップ410において内部記号表75から検索により取り出されたそれぞれの次元についての許容最大サイズとともに、「dimchk」コマンドノードをステップ425においてパーズ木に挿入することが好ましい。「dimchk」コマンドノードは、次元参照が配列のこの次元についての有効な領域範囲内にあるかどうかを定めるための評価が実行時において行われる。

【0094】「dimchk」コマンドノードがパーズ木に挿入されてしまうと、手順はステップ450に進みここで、この配列に検査すべき他の次元があるかどうかの上に述べた方法で判断され定められる。配列の次元がすべて検査されてしまうと、手順はステップ320（図7）において「読み出し時エラー検査プロセス」に戻る。

【0095】もし「読み出し時エラー検査プロセス」によって評価中の原始コードファイルのラインが、ポインタに値を割り当てることがステップ320（図7）において検出された場合、手順はステップ505（図9）において図9および図10に示す「読み出し時ポインタ検査表更新サブルーチン」に入り、同サブルーチンが開始される。

【0096】ポインタ検査表200がそれぞれのポインタについての新しい情報によって実行時において適切に更新されるように、ポインタに値が割り当てられる都度、「読み出し時ポインタ検査表更新サブルーチン」が、必要なコマンドおよび情報をパーズ木に挿入する。

【0097】「読み出し時ポインタ検査表更新サブルーチン」が開始されると、ステップ505において、「tblupd」コマンドノードがパーズ木に挿入される。

「tblupd」コマンドノードは、実行時において評価されると、ポインタ検査表200の更新を開始する。実行時におけるポインタ検査表更新を行うために、読み出し時において取得できる追加情報、すなわち、ポインタに割り当てられるポインタ式の型の特性が、「tblupd」コマンドノードに入れられる。

【0098】そして、ポインタに割り当てられるポインタ式が、ステップ510において解析され、ステップ515からステップ560（図10）までにおいて、型のマッチングが得られるまで、いくつかの条件に対して検査される。

【0099】ステップ515において、ポインタが、識別特定された変数または関数のアドレスを割り当てられているかどうかを定める検査が行われる。もしステップ515において、ポインタが、識別特定された変数または関数のアドレスを割り当てられていると判断された場合、ステップ518において、特定された変数または関数についての内部記号表75内の適切なエントリを指示するポインタが、「tblupd」コマンドノードに入れられる。

【0100】実行時において、内部記号表75内のこのエントリが、必要なポインタ情報を得るためにアクセスされる。その後、手順はステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0101】もしステップ515において、ポインタが、識別特定された変数または関数のアドレスを割り当てられていないと判断された場合には、手順はステップ520に進む。

【0102】ステップ520において、ポインタが、第2のポインタの内容を割り当てられているかどうかを定める検査が行われる。もしステップ520において、ポインタが、第2のポインタの内容を割り当てられていると判断された場合、手順はステップ522に進む。この場合、実行時において第2のポインタが何を指示するか、および第2のポインタの有効な領域の範囲はどこかを読み出し時において判断することは不可能である。

【0103】しかし、実行時において、第2のポインタのアドレスはインタプリタスタック60の最上部に位置する。したがって、ステップ522において、「readintstk」コマンドが「tblupd」ノード内に入れられる。「readintstk」コマンドは、実行時において評価されると、第2のポインタのアドレスを、さらに処理するために、インタプリタスタック60から検索により取り出させる。その後、手順はステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0104】もしステップ520（図9）において、ポインタが、第2のポインタの内容を割り当てられていないと判断された場合、手順はステップ525に進む。

【0105】ステップ525において、ポインタが、記号列（ストリング）を割り当てられているかどうかを定める検査が行われる。もしステップ525において、ポインタが、ストリングを割り当てられていると判断された場合、割当を受けようとするポインタについてポインタ検査表200の適切な横列に配置する必要がある情報、すなわちストリングのアドレスおよびサイズ、は読み出し時において既知である。

【0106】そして、ステップ527において、ストリングのアドレスおよびサイズが「tblupd」コマンドノード内に入れられる。その後、手順はステップ330（図7）において「読み出し時エラー検査プロセス」

に戻る。

【0107】もしステップ525（図9）において、ポインタが、ストリングを割り当てられていないと判断された場合、手順はステップ530に進む。

【0108】ステップ530において、ポインタが、不正値を割り当てられているかどうかを定める検査が行われる。もしステップ530において、ポインタが、不正値、すなわち定常的なゼロ、または負の値、を割り当てられていると判断された場合、ステップ532において、「不正」状態の表示が「tblupd」ノード内に
10 入れられる。ポインタは不正値を記憶することが許容されるので、エラーメッセージはここでは生成されない。

【0109】しかし下でさらに述べるように、もしポインタが実行時において参照を除去され、しかもなお不正値を有する場合には、その時点においてエラーメッセージが生成される。ステップ532の実行に続いて、手順はステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0110】もしステップ530（図9）において、ポインタが、不正値を割り当てられていないと判断された
20 場合、手順はステップ535に進む。

【0111】ステップ535において、ポインタが、構造体メンバのアドレスを割り当てられているかどうかを定める検査が行われる。もしステップ535において、ポインタが、構造体メンバのアドレスを割り当てられていると判断された場合、構造体メンバのサイズは読み出し時において既知である。しかし、構造体メンバのアドレスは、実行時になるまで不明のままであり、実行時において構造体メンバのアドレスは、インタプリタスタック60の最上部に位置する。

【0112】そして、ステップ537において、構造体メンバのサイズおよび「readintstk」コマンドが「tblupd」ノード内に入れられる。「readintstk」コマンドが実行時において評価されると、構造体メンバのアドレスがインタプリタスタック60から検索により取り出される。ステップ537の実行に続いて、手順はステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0113】もしステップ535（図9）において、ポインタが、構造体メンバのアドレスを割り当てられてい
40 ないと判断された場合、手順はステップ540（図10）に進む。

【0114】ステップ540においてポインタが、複合式、例えば $p = * (test?exp1:exp2)$ によって割り当てられているかどうかを定める検査が行われる。ここに、 $exp1$ および $exp2$ はポインタのアドレスを評価するポインタ式である。この特殊検査条件は、「if, then, else」ルーチン実現用のこの普通のプログラミング手法についてエラー検査を行うために実現される。

【0115】間接演算子「*」は、検査条件が真かまたは偽りにかいて $exp1$ または $exp2$ のいずれかについて行われるので、これら2式のうちのどちらが新しいポインタの特性を定めるかについては、読み出し時においては未知である。

【0116】本発明の好ましい一実施例において、「savptr」コマンドのような命令が、2個の命令ストリームの各々に入れられる。実行時において実行されるのは、これら2個のうちの1個だけである。「savptr」コマンドは、実行された命令に関連するポインタについてのポインタ情報をポインタ保存スタック65に入れるように評価器35に命令する。このポインタ情報は、次に評価器35からアクセスされる。

【0117】ポインタ保存スタック65から検索により取り出されたポインタ情報は、ポインタ検査表200に入れるため、実行時において評価器35によって、下にさらに述べる方法で検索により取り出すことができる。

【0118】そして、もしステップ540において、ポインタが形式「 $*(test?exp1:exp2)$ 」の条件式によって割り当てられていると判断された場合には、ステップ542において、「savptr」コマンドが生成され、2個の式 $exp1$ および $exp2$ に関連する命令ストリームの各々に入れられる。

【0119】これに加えて、「rdptrstk」コマンドのような、第2の命令が「tblupd」ノードに入れられる。この第2の命令は、評価されると、評価器35をしてポインタ保存スタック65の内容を検索により取り出させる。ステップ542の実行に続いて、手順はステップ330（図7）において「読み出し時エラー
30 検査プロセス」に戻る。

【0120】もしステップ540（図10）において、ポインタが、このような複合式によって割り当てられていないと判断された場合には、手順はステップ545に進む。

【0121】ステップ545において、ポインタが、翻訳された関数への1個以上のポインタ引数の送付後すぐに暗黙のうちに割り当てられるかどうかを定める検査が行われる。プログラムが、翻訳された関数を呼び出すとき、「呼び出し」命令文に規定されるような、呼び出された関数の引数、すなわち実引数が、宣言された関数内のそれぞれの自動変数、すなわち形式引数に暗黙のうちに割り当てられる。

【0122】関数が呼び出されると、評価器35が、呼び出し関数についてメモリ割付を受けているハードウェアスタック67に、呼び出された関数の引数を押し込む。その後、呼び出された関数のための新しいスタックフレームが、評価器35により、スタックフレームポインタを更新することによって形成される。なお、呼び出された関数によって用いられるスタックフレームは、呼
50 び出し関数のスタックフレームのアドレスとは異なるス

タックアドレスを持つことになる。

【0123】形式パラメータについてのポインタ検査表200内のエントリは、その関数へ送られた実ポインタパラメータ、すなわちポインタ、についてのポインタ検査表200内のエントリをコピーすることによって形成するのが好ましい。すなわち、ポインタである送られた実パラメータについてのポインタ検査表200からのポインタ情報がポインタ保存スタック65に入れられるのが好ましい。

【0124】このようにして、評価器35は、ポインタ保存スタック65内に入れられたポインタ情報をコピーすることによって、ポインタである型パラメータについて実行時においてポインタ検査表200内に適切なエントリを形成することができる。

【0125】そして、もしステップ545において、翻訳された関数へのポインタ引数の送付後すぐに暗黙のうちに割り当てられると判断される場合、手順はステップ547に進む。ここで、各ポインタ引数についての関数呼び出しに先立ち、「savptr」コマンドがパズ木の中に入れられる。

【0126】本発明の好ましい実施例において、「savptr」コマンドは、送られたポインタ引数を受け取ることになる呼び出された「翻訳された関数」を指示するポインタを含む。「翻訳された関数」を指示するポインタは、ポインタ保存スタック65内に入れられた情報の有効性を確実なものにするための識別子の役をする。これに加えて、「rdptrstk」コマンドが「tblupd」ノードに挿入される。

【0127】「savptr」コマンドが実行時において次に評価されるとき、ポインタである「送られた実パラメータ」についてのポインタ情報がポインタ保存スタック65に入れられる。その後、実行時における「rdptrstk」コマンドの評価が、評価器35をして、ポインタ保存スタック65の内容を検索により取り出させる。この内容は、ポインタ検査表200に入れられる。ステップ547の実行に続いて、手順はステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0128】もしステップ545において、翻訳された関数へのポインタ引数の送付後すぐに暗黙のうちに割り当てられることがないと判断される場合、手順はステップ550に進む。

【0129】ステップ550において、ポインタが、翻訳された関数から返された値を割り当てられるかどうかを定めるテストが行われる。読み出し時において関数から戻されることになる値についての情報は、もしあったとしても少ししかないので、返されたポインタについてのポインタ情報を実行時においてポインタ保存スタック65に押し込む命令がパズ木に入れられる。

【0130】これによって、評価器35が、返されたポ

インタに関連するポインタ情報を、ポインタ検査表200に入れるために実行時においてポインタ保存スタック65から次に検索により取り出すことができる。

【0131】そして、もしステップ550において、ポインタが、翻訳された関数から返された値を割り当てられていると判断された場合、ステップ552において、呼び出された関数からの返呼に応答して、返されたポインタの各々について「savptr」コマンドが生成される。これに加えて、「rdptrstk」コマンドが、「tblupd」ノードに入れられる。

【0132】本発明の好ましい実施例において「savptr」コマンドは、ポインタ情報を返すポインタである「翻訳された関数」を指示するポインタを含む。この「翻訳された関数」を指示するポインタは、ポインタ保存スタック65内に入れられた情報の有効性を確実なものにするための識別子の役をする。その後、手順はステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0133】「savptr」コマンドが実行時において次に評価されるとき、関数から戻されるポインタに関連するポインタ情報がポインタ保存スタック65に入れられる。その後、実行時における「rdptrstk」コマンドの評価が、評価器35をして、ポインタ保存スタック65の内容を検索により取り出させる。この内容は、ポインタ検査表200に入れられる。

【0134】なお、関数がポインタを自動変数に返すことは不適切である。したがって、この場合にはエラーが発生する。

【0135】もしステップ550（図10）において、ポインタが、翻訳された関数から戻された値を割り当てられていないと判断された場合、手順はステップ560に進む。

【0136】もし手順がステップ560に到達した場合、ポインタに割り当てられているポインタ式は、上に設定されたテストのうちのどれか1つに基づく特徴付けができないことになる。すなわち、読み出し時においては、割り当てられるポインタについて何も確定せず、したがってステップ560において、「不明」状態の表示が「tblupd」ノードに入れられる。ステップ560の実行に続いて、手順は、ステップ330（図7）において「読み出し時エラー検査プロセス」に戻る。

【0137】もし「読み出し時エラー検査プロセス」によって評価中の原始コードファイルのラインが、少なくとも1回のポインタ関連解除を含むことがステップ330（図7）において検出された場合、手順は図11のステップ615において、図11に示す「読み出し時ポインタ検査サブルーチン」に入り、同サブルーチンが開始される。関連解除されたポインタの有効性をテストするために実行時においてポインタ検査表200にアクセスできるように、ポインタが関連解除される都度、「読み

出し時ポインタ検査サブルーチン」が、必要なポインタ検査コマンドをパズ木に挿入する。

【0138】ステップ615において「読み出し時ポインタ検査サブルーチン」が開始されると、ステップ615において、関連解除されるポインタについて「ptrchk」コマンドノードがパズ木に挿入される。関連解除されるポインタの値は、ポインタの値が計算された後でメモリにアドレスするのにこの計算された値が用いられる前にはインタプリタスタック60で最も上部にある項目なので、「ptrchk」コマンドノードは、メモリアドレス命令の直前位置に入れられる。

【0139】「ptrchk」コマンドノードが実行時において評価されると、図18および図19を参照して下で説明する「実行時ポインタ検査サブルーチン」が開始され、インタプリタスタック60から検索により取り出された値が、それぞれのポインタについて、ポインタ検査表200の有効なメモリ境界エントリ250a、250bに示されるようなそのポインタの有効領域範囲内にあるかどうか判断される。

【0140】ステップ620において、評価中のパズ木のラインに他のポインタ関連解除があるかどうかを定めるテストが行われる。もしステップ620において、評価中のパズ木のラインに他のポインタ関連解除があると判断された場合、手順はステップ615に戻り、そこでさらに処理が続けられる。ステップ620において、評価中のパズ木のラインにあるポインタ関連解除のすべてが処理されたと判断された場合、手順はステップ335（図7）において「読み出し時エラー検査プロセス」に戻る。

【0141】ステップ335においては、「読み出し時エラー検査プロセス」によって評価中の原始コードファイルのラインが関数への呼び出しを含むかどうかを判断するテストが行われる。もしステップ335において、評価中の原始コードファイルのラインが関数への呼び出しを含むと判断された場合、その関数によってアクセスされるメモリのすべてが、ステップ337において識別特定される。すなわち大域変数のすべておよびその関数に送られた変数のすべてである。

【0142】これに加えて、もしその関数がメモリにアクセスを有する場合には、メモリ割付アクセスフラグが、コンパイルされた関数に対応する内部記号表75内のエントリにセットされる。図21を参照して下に述べるように、このメモリ割付アクセスフラグをセットする動作が、連関する関数が実行時において呼び出される都度、「初期化ビットベクトル維持サブルーチン」を起動させる。

【0143】もしステップ335において、評価中の原始コードファイルのラインが関数への呼び出しを含まないと判断された場合、手順はステップ340に進む。

【0144】ステップ340においては、「呼び出し時

エラー検査プロセス」が、まだ読み出され解析される必要のあるパズ木の他のノードがあるかどうかを判断するテストを行う。もしステップ340において、まだ読み出され解析される必要のあるパズ木の他のノードがあると判断された場合、手順はステップ300に戻り、上に述べたように進む。しかし、もしステップ340において、まだ読み出され解析されるべきパズ木の他のノードが残っていないと判断された場合、手順はステップ350に進む。

【0145】ステップ350においては、メモリアクセスエラー検出コード挿入器25によって、修正パズ木についてデータ流れ解析が行われる。修正パズ木には、パズ木に挿入されたエラー検査コマンドおよび追加情報を含む。データ流れ解析は、パズ木についての重複すなわち重合するポインタ検査を削除することによって、実行時において行う必要のあるポインタ検査の数を最小化し、ある読み出し時エラーを検出できるようにする。

【0146】適切なデータ流れ解析アルゴリズムの記述については、W・ランディおよびB・ライダー（William Landi & Barbara G. Ryder）の論文「ポインタのある場合およびない場合のエーリアシング：問題分類学」（Center for Computer Aids for Industrial Productivity, Technical Report CAIP-125、ラトガーズ大学、1990年9月25日）を参照されたい。

【0147】また同じく、W・ランディおよびB・ライダー（William Landi & Barbara G. Ryder）の論文「手続き間相互ポインタエーリアシングについての安全近似アルゴリズム」（SIGPLAN Notices、1992年7月、235～248ページ）を参照されたい。

【0148】図23に示す原始コードファイル例「サンプル」のデータ流れ解析について下に簡単に述べる。データ流れ解析は、ポインタが新たな値を割り当てられまたは関連解除される都度、修正パズ木を解析し識別特定する。なお、「tblupd」コマンドノードは前に、ポインタ割当の各々についてパズ木に入れられており、「ptrchk」コマンドノードは前に、ポインタが関連解除される都度パズ木に挿入されている。

【0149】図23に示す原始コードファイル「サンプル」は、構造体「widget」を指示するポインタ「ptr_part」を宣言し、初期化する。データ流れ解析は、評価中の原始コードファイルに現れる各ポインタについて流れセットを設定する。この流れセットは例えば、原始コードファイル「サンプル」の実行時に形成されたポインタ「ptr_part」について図24に示すような、流れセットである。

【0150】与えられたポインタに対して、その連関するポインタが新たな値を割り当てられる都度、新たな流れセットが設定される。好ましい実施例においては、流れセットは、その連関するポインタが正規の値を割り当

てられる都度「DEF」とマーク付け、すなわち定義され、その関連するポインタが不正値を割り当てられる都度「DEF ILL」とマーク付けされる。同様に、その関連するポインタが関連解除される都度、「USED」（使用済）とマーク付けされる。

【0151】図23に示す原始コードファイル「サンプル」のライン10からライン50までは、構造体「widget」を宣言する。ライン60は構造体「widget」を指示するポインタ「ptr_part」を宣言し、ライン70は構造体「widget」の始点を指示するためにこのポインタを初期化、すなわち定義する。ライン70のデータ流れ解析によって、図24に示すこのポインタ「ptr_part」についての第1の流れセットが「DEF」とマーク付けされる。このマーク付けは、このポインタが「非ゼロ」値を割り当てられたことを表す。

【0152】構造体「widget」の3個のメンバに値を割り当てるために、ポインタ「ptr_part」がライン80からライン100までの各ラインにおいて関連解除される。そして、ライン80からライン100までの各ラインのデータ流れ解析によって、図24における第1の流れセットが「USED」とマーク付けされる。このマーク付けは、このポインタが関連解除されたことを表す。

【0153】それからポインタ「ptr_part」がライン150において新たな値を割り当てられ、これによって、ポインタ「ptr_part」についての新たな流れセットが設定される。そして、ライン150のデータ流れ解析によって、図24に示す第2の流れセットが「DEF ILL」とマーク付けされる。このマーク付けは、このポインタが不正値を割り当てられたことを表す。

【0154】そして、構造体「widget」のメンバ「number」に値を割り当てるために、ポインタ「ptr_part」がライン160において関連解除される。また、ライン160のデータ流れ解析によって、図24における流れセットが「USED」とマーク付けされる。このマーク付けは、このポインタが関連解除されたことを表す。下で説明するように、この第2の流れセットの解析によって、読み出し時エラーが発生する。その理由は、ライン160が実行されるときに不正ポインタが関連解除中だからである。

【0155】パズ木全体について流れセットが設定されると、実行時において行う必要のあるポインタ検査の数を最小化するために、各流れセットが解析される。定義によって、与えられたポインタは各流れセットの期間中同一の値を維持するので、各セットについて、各々の個別ポインタ検査を受け入れ可能なようにオフセットを拡張したポインタ検査を1回だけ行えばよい。

【0156】例えば、ポインタ「ptr_part」に

についての第1の流れセットには、このポインタの3回の関連解除が含まれる。すなわち、ポインタはライン70において割り当てられた値を記憶している一方、関連解除は3回行われることになる。通常、データ流れ解析を行わない場合には、3回の別個のポインタ検査を行う必要がある。しかし、データ流れ解析によって、パズ木における3回の別個のポインタ検査が、3回のポインタ関連解除の各々の有効性をテストする1回だけのポインタ検査に置換される。

【0157】本発明の好ましい実施例においては、上記データ流れ解析の正確さを確実なものにするための「コンパイルされたコード流れ解析ルーチン」が設けられ、このルーチンにおいては、与えられたポインタについての流れセットに、一連のポインタ関連解除とともに、コンパイルされた関数の呼び出しが含まれる。もしこのコンパイルされた関数が、この流れセットに関連するポインタへのアクセスを有する場合、このコンパイルされた関数は、翻訳器15に気付かれずにこのポインタの値を修正できる。

【0158】そして、もしこのコンパイルされた関数がこのポインタへのアクセスを有する場合、実行時におけるコンパイルされた関数の実行後に、コンパイルされたコードの流れ解析ルーチンが、この流れセットに関連するポインタ値がコンパイルされた関数の実行中に修正されているかどうかを判断するテストを行うのが好ましい。

【0159】もしコンパイルされた関数がポインタ値を修正していた場合、このコンパイルされた関数の呼び出し後流れセットにおいて生じるこれらのポインタ関連解除についてポインタ検査を追加して再度実行する必要がある。もしコンパイルされた関数がポインタ値を修正していない場合、ポインタ検査を追加して行う必要はない。

【0160】実行時において行うべきポインタ検査の数を最小化することに加えて、上記のデータ流れ解析によって、あるエラーを読み出し時において検出することが可能となる。すなわち、ステップ350におけるデータ流れ解析の完了後、ステップ360において、初期化されていないまたは不正なポインタが関連解除されつつあることをこのデータ流れ解析が表示しているかどうかを判断するテストが行われる。

【0161】初期化されていないポインタを関連解除するというメモリエラーは、ポインタが割り当てられる前に関連解除されることを流れセットが表示する場合、すなわち流れセットが「DEF」とマーク付けされる前に「USED」とマーク付けされる場合に生じる。同様に、不正なポインタを関連解除するというメモリエラーは、図24の第2の流れセットにおけるように、流れセットが「DEF ILL」とマーク付けされ、次に中間に来る割当なしに、「USED」とマーク付けされる場

合に生じる。

【0162】もしステップ360において、不正なまたは初期化されていないポインタが関連解除されていると判断された場合、ステップ370においてエラーメッセージが生成され、その後、手順はステップ380においてプロセスから出て、プロセスが終了する。

【0163】しかし、もしステップ360において、不正なまたは初期化されていないポインタが関連解除されていないと判断された場合、手順はステップ380において「読み出し時エラー検査プロセス」から出て、プロセスが終了する。

【0164】図7から図11までを参照して上に述べた「読み出し時エラー検査プロセス」および連関するサブルーチンが完了すると、パズ木の各ラインの解析が終り、適切なエラー検査コマンドおよび情報のパズ木への挿入が終る。そして、実行時エラー検査を起動実行するために、挿入されたエラー検査コマンドおよび追加情報が評価されることになる。

【0165】「コンパイルされた目的コードのエラー検査」本発明の別の特徴によれば、実行時において評価器35によって実行される、「コンパイルされた目的コード」についての同様なエラー検査を行うことも望ましい。

【0166】本発明の好ましい実施例において、「コンパイルされた目的コードのエラー検査」は、読み出し時において「翻訳された原始コード」について行われたエラー検査プロセスをまねるように、連関する「コンパイルされた関数」の実行時における実行の、必要に応じて前および／または後に、下に述べるように「翻訳されたラップ関数」を実行することによって実現される。

【0167】好ましくは、すべての「コンパイルされたライブラリ関数」について、必要に応じてラップ関数を設ける。これに加えて、ラップ関数は、ポインタ値の監視および／または評価を要する、ユーザにより定義されるどの「コンパイルされた関数」、についても本明細書における教示に基づいて形成することができる。実行時における、連関する「コンパイルされた関数」の呼び出し時にラップ関数が実行される実行方法について図20を参照して下に説明する。

【0168】評価器35によってアクセス可能なラップライブラリ70を図1に示すように設けるのが好ましい。ラップライブラリ70は、「コンパイルされたコード」の実行中にポインタを関連解除する「コンパイルされた関数」の各々について「実行前ラップ関数」を維持することが好ましい。実行前ラップ関数は、「コンパイルされたコード」の実行中に関連解除されるポインタのポインタ検査を、「翻訳された関数」においてポインタの関連解除のためにポインタ検査が行われるのと同じ方法で、起動する必要がある。

【0169】「実行前ラップ関数」は、連関する「コン

パイルされた関数」の実行の直前に実行するのが好ましい。これによって、ポインタが有効値を指示していなければそのポインタは関連解除されないということが確実になる。与えられた関数についての実行前ラップ関数はまた、配列参照を含む送られた引数について、ここに述べるように配列次元検査をも行うことが好ましい。

【0170】同様に、もし与えられた「コンパイルされた関数」へ送られた引数が予め定義された有効範囲を有する場合、連関する「実行前ラップ関数」は、この既知の範囲に対する送られた引数の有効性をテストすることができる。

【0171】なお、ある「コンパイルされた関数」については、連関する「実行前ラップ関数」は、「コンパイルされた関数」の第1の実行時に「コンパイルされた関数」へ送られたポインタ引数またはその他の情報を記憶保存する必要がある。一方、同一関数への次の呼び出しは、前の実行時と同じ情報を用いる必要があることを表示するコードを送るだけである。

【0172】これに加えて、ラップライブラリ70は、「コンパイルされたコード」の実行中にポインタを形成する「コンパイルされた関数」の各々について、および呼び出し関数に戻るときに次にポインタに割り当てられる値を返す「コンパイルされた関数」の各々について、「実行後ラップ関数」を維持することが好ましい。「実行後ラップ関数」は、「コンパイルされた関数」に戻る前に、しかし呼び出し関数が実行を再開する前に、実行することが好ましい。

【0173】もし「コンパイルされた関数」が、実行中にポインタを形成する場合には、「実行後ラップ関数」は、その形成されたポインタについてのポインタ情報をポインタ検査表200に追加する必要がある、この追加は、ポインタ検査表エントリが「翻訳されたコード」によって形成されたポインタについて更新されるのと同じ方法で行う必要がある。

【0174】同様に、もし「コンパイルされた関数」が、次に呼び出し関数においてポインタに割り当てられることになる値を呼び出し関数に返す場合には、適切なポインタ情報をポインタ検査表200に追加する必要がある。この場合、返される値についてのポインタ情報をポインタ保存スタック65に、読み出し時ポインタ検査表更新サブルーチンがステップ552（図10）において「翻訳された関数」から返される値を取り扱ったのと同じ方法で入れる必要がある。

【0175】この後、ポインタ検査表200に入れる目的で、この情報がポインタ保存スタック65から評価器35によって検索により取り出される。

【0176】例えば、ポインタには、Cプログラミング言語の関数ライブラリに普通に見出される「malloc」関数のような、コンパイルされたメモリ割付関数から戻される値が頻繁に割り当てられる。「malloc

c) 関数についての「実行後ラッパ関数」は、割付されたメモリ領域の始点アドレスと、始点アドレスおよびサイズ情報から導かれるこの領域の有効範囲とをポインタ保存スタック65に入れることが好ましい。

【0177】このようなふうにして、返された値が呼び出し関数におけるポインタに割り当てられるとき、ポインタ検査表200(図2)における適切な横列が、関連するポインタ情報データ更新される。始点アドレスがポインタエントリ240の内容欄に入れられ、有効範囲エントリ250には、「実行後ラッパ関数」によって計算された範囲情報が入れられる。好ましくは、「割付済」の状態表示を状態エントリ260に入れるようにする。

【0178】しかし、本発明の好ましい実施例によれば、割付されたメモリスペースを指示するポインタについて追加ポインタ情報が維持される。図3に示す原始コードファイル例「テストファイル」のライン200およびライン210が実行されると、35バイトのブロックが割付される結果となり、割付されたブロックの始点アドレスがポインタ「ptr_allc」に割り当てられる。

【0179】「malloc」関数から戻ると、ポインタ「ptr_allc」に対するポインタ検査表200内の横列が、新たなポインタ情報を反映するように更新される。ポインタアドレスエントリ230とポインタ内容エントリ240には、適切な情報が入れられる。状態エントリ260には、割付されたメモリスペースをポインタが指示することを表す表示「割付済」(ALLOCATED)が入れられる。

【0180】この好ましい実施例においては、「malloc」関数についての「実行後ラッパ関数」もまた、図6に示すようなメモリ割付構造体280を形成する。図2に示すように、ポインタ「ptr_allc」についての有効下側メモリ境界250aはメモリ割付構造体280を指示するポインタを有する。

【0181】メモリ割付構造体280は、下側境界部分282、上側境界部分284、チェインリスト部分286、初期化ビットベクトル288、および状態部分290からなる。メモリ割付の有効上側および下側境界は、構造体部分282および284にそれぞれ保存記憶される。

【0182】チェインリスト部分286は、割付されたメモリの関連ブロックを現に指示するすべてのポインタのリストからなる。割付されたメモリを指示するポインタが他のポインタにコピーされる都度、第2のポインタに対するポインタ検査表200内の状態エントリ260も「割付済」とマーク付けされ、有効下側境界エントリ250a内の同じメモリ割付構造体280を指示するポインタを有する。これに加えて、新たなポインタがチェインリスト部分286内のリストに追加される。

【0183】同様に、もし割付されたメモリを指示する

ポインタが、割付されたメモリの新たなブロックを指示するように指示について再割当された場合、このポインタを新たなメモリ割付についてのチェインリスト部分286に追加する前に、前回のメモリ割付についてのチェインリスト286内のリストからこのポインタを除去することが好ましい。下でさらに述べるように、本発明のこの特徴により、メモリ漏洩を自動的に検出することが可能となる。

【0184】結果として、メモリ割付構造体280のチェインリスト部分286に記録されている情報は、連関する割付されたメモリを現に指示するポインタだけを表示する。

【0185】図6に示すように、メモリ割付構造体280に含まれる初期化ビットベクトル288は、割付されたメモリの各バイト当たり1個のビットからなり、連関するメモリの各バイトの初期化状態を維持表示する。本発明の好ましい実施例において、割付されたメモリの領域内の全バイトが初期化され終ると、初期化ビットベクトル288は廃棄され、この領域全体の初期化状態を表示するフラグがセットされる。

【0186】下でさらに述べるように、割付されたメモリからの読み出しがポインタを用いて行われる都度、割付されたメモリのそれぞれのバイトが初期化されたことを確実にするために、初期化ビットベクトル288が評価される。これに加えて、下でさらに述べるように、「コンパイルされたコード」または「翻訳されたコード」のいずれかによって割付されたメモリの初期化が検出された後に初期化ビットベクトル288の適切なビットを更新するためのメカニズムが設けられる。

【0187】本発明の好ましい実施例においては、「malloc」関数のようなメモリ割付関数、に連関する「実行後ラッパ関数」が、大抵のプログラミング実例では出会いそうにないような既知の予め定義された値をこの関数によって割付されているメモリの4バイト領域、の各々について事前のマーク付けを行うのが好ましい。この事前マーク付けされた値によって、メモリ領域の初期化の検出が可能となる。

【0188】プログラミング環境では16進値「FFFA 5A5A」には次の2つの理由から通常出会わないことが判っている。第1の理由は、もしこの値が浮動小数点値として用いられる場合この値がIEEE浮動小数点基準において「非番号」として定義されるので、浮動小数点トラップエラーが発生するためである。第2の理由は、もしプログラマがこの値をポインタとして用いようとする場合、大抵のマシンにセグメンテーションまたはメモリ欠陥エラーが発生するためである。

【0189】すべての動的に割付されたメモリを16進値「FFFA 5A5A」で事前マーク付けすることに加えて、すべての初期化されていない自動変数に連関するメモリ位置が実行時において同様にマーク付けされる

こともまた好ましい。なお、これに加えて、すべての初期化されていない静的変数は、実行時においてリンク装置40により「0」にセットされる。

【0190】「malloc」関数のようなメモリ割付関数に通常連関する動作速度は遅いので、多くのプログラムはしばしば、1個の大きなメモリブロックの割付を行い、それからその大きなブロックを、必要に応じてより小さい部分に分解し、割付を受けたこれらのより小さい部分の各々を少なくとも1個のポインタでアクセス可能のようにしている。しかし、上に概略説明したポインタ検査表更新プロセスによれば、各ポインタの有効領域範囲は通常、大きなメモリブロック全体としてポインタ検査表200に記録されることになる。

【0191】すなわち、ポインタ検査表200は、より大きなメモリ割付分のうちのより小さな部分だけを指示するように意図されているポインタについての適切な領域範囲を表示する必要がある。したがって、より小さい領域範囲を記録するために、ポインタ検査表200内のエントリにアクセスできることが好ましい。

【0192】本発明のさらに別の特徴によれば、割付されたメモリの割付解除（freeライブラリ関数を呼び出すことによって行われる）も図12に示す「メモリ割付解除をモニタするためのサブルーチン」によってモニタされる。メモリ割付解除関数が呼び出されると、メモリ割付解除関数の引数は一般に、割付されたメモリスペースを指示するポインタのうちの1個である。

【0193】共通するプログラミングエラーの1つは、ある1個のポインタを用いてメモリを割付解除し、それから同じメモリスペースを、その同じメモリスペースを指示するように前に定義されていた別個のポインタでアクセスしようと意図する場合である。

【0194】連関するコンパイルされたメモリ割付解除関数がステップ737において呼び出され実行される前に、「メモリ割付解除をモニタするためのサブルーチン」のステップ705からステップ735までにおいて「実行後ラップ関数」が起動されるのが好ましい。「実行後ラップ関数」は、解除された（freeされた）メモリにアクセスしようと意図するときに引き続いてエラーが発生するのを防止する。

【0195】「メモリ割付解除をモニタするためのサブルーチン」は、メモリ割付解除関数へ送られたポインタ引数について、ステップ705においてポインタ検査表200内の横列を最初に探索する。

【0196】ステップ710において、ポインタが「解放」（FREED）状態にあることを状態エントリ260が表示しているかどうかを定めるテストが行われる。もしステップ710において、状態が「解放」であると判断された場合、ステップ715においてエラーメッセージが生成される。その理由は、そのポインタによって指示された割付されたメモリが既に解放されているから

である。その後、手順はステップ740においてサブルーチンから出て、サブルーチンが終了する。

【0197】ステップ720において、状態エントリ260が「メモリ割付済」の状態または「不明」の状態かどうかを定めるテストが、「メモリ割付解除をモニタするためのサブルーチン」によって行われる（状態エントリ260（図2）および状態部分290（図6）において、「メモリ割付済」の状態および「不明」の状態をそれぞれ「割付済」および「不明」とマーク付けする）。

【0198】もしステップ720において、状態が「メモリ割付済」の状態または「不明」の状態のいずれでもないと判断された場合、ステップ725においてエラーメッセージが生成される。その理由は、これらの状態だけが、割付されたメモリについての有効な状態コードだからである。その後、手順はステップ740においてサブルーチンから出て、サブルーチンが終了する。

【0199】もしステップ720において、状態が「メモリ割付済」の状態または「不明」の状態であると判断された場合、「メモリ割付解除をモニタするためのサブルーチン」がステップ727において、下側境界エントリ250a内でそのポインタによって指示されるメモリ割付構造体280にアクセスし、「解放」の状態を表示するように状態部分290（図6）をセットする。

【0200】その後、ステップ730において、「メモリ割付解除をモニタするためのサブルーチン」が、下側境界エントリ250a内でそのポインタによって指示されるメモリ割付構造体280にアクセスし、割付されたメモリスペースを指示するすべてのポインタのリストを検索により取り出す。

【0201】ステップ735において、前のステップにおいて検索により取り出されたリストに表示される各ポインタについて、ポインタ検査表200の適切な横列がアクセスされ、状態エントリ260が「解放」とマーク付けされる。

【0202】これに加えて、ステップ737におけるコンパイルされたメモリ割付解除関数の実行に続いて、ステップ739において、「実行後ラップ関数」がリストに入れられた各ポインタの実際の内容をゼロ値にセットするのが好ましい。この場合、もしゼロ値を有するこれらのポインタのうちの1個を関連解除する試みがなされると、エラーメッセージが生成される。この後、ステップ740において手順はサブルーチンを出て、サブルーチンが終了する。

【0203】なお、翻訳された関数においてローカル変数について形成されたメモリスペースを静的ポインタが指示すると、暗黙のメモリ割付解除が生じる。翻訳された関数が戻ると、割付されたメモリスペースは、翻訳器15によって自動的に割付解除される。

【0204】静的ポインタは、関数からの戻りに続いて、無効メモリスペースを指示するので、静的ポインタ

の値をゼロ値にセットすることが好ましく、また、静的ポインタについてのポインタ検査表200内の横列が、「0」の内容を表示するように更新されるのが好ましい。これは、静的ポインタにローカル変数のアドレスを割り当てた「翻訳された関数」から戻ったときに図12の「メモリ割付解除をモニタするためのサブルーチン」を実行することによって実現される。

【0205】「実行時エラー検査動作」評価器35が実行時において内部疑似コードの各ノードを評価すると、図13に示すような、そのノードが、エラー検査ルーチンに関連するコマンドを有するかどうかを定める「実行時エラー検査プロセス」が実現される。「実行時エラー検査プロセス」は、図7を参照して上に述べた「読み出し時エラー検査プロセス」によってパズ木に挿入されたあるエラー検査コマンドおよび追加情報に応答して、内部疑似コードの各ノードを評価し、実行時エラー検査タスクを実現する。

【0206】「実行時エラー検査プロセス」は、図13に示すように、評価中のノードが実行時エラー検査ルーチンに関連する5個のコマンドのうちの1個を有するかどうかを判断する。もしエラー検査コマンドが検出された場合、「実行時エラー検査プロセス」が、図13に示すように、適切な応答を起動する。

【0207】ステップ800において手順が「実行時エラー検査プロセス」に入り、ここで内部疑似コードの各ノードが評価された後、もしステップ810において「dimchk」コマンドが存在することが検出されると、図14を参照して下に述べる「実行時配列次元検査サブルーチン」が実行される。

【0208】「実行時配列次元検査サブルーチン」は、実行時において配列が、不正な添字を参照として付けられているかどうかを定める。ステップ910（図14）において、変数配列添字が、実行時条件に基づいて計算される。その後、ステップ920において、計算された添字参照が負数かまたは有効最大次元を超えているかどうかを定めるテストが行われる。なお、有効最大次元は、読み出し時において計算され、「dimchk」コマンドに入れられたものである。

【0209】もしステップ920において、計算された添字参照が不正値であると判断された場合、ステップ924において、有効最大次元が「1」であるかどうかを定めるテストが行われる。

【0210】もしステップ924において、有効最大次元が「1」であると判断された場合、本明細書に開示されているポインタ検査手段がステップ928において用いられ、構造体が全体として有効範囲内にあるかどうか定められる。単一要素を有する配列で構造体宣言を終り次いで実行時における配列の長さを選択するという共通のプログラミング手法が、このステップ924において行われるテストによって可能になる。なお、構造体

が割付されるときには、配列の追加構成部分に対して追加のメモリスペースが割付される。

【0211】もしステップ924において、有効最大次元が「1」ではないと判断された場合、エラーは、この共通プログラミング手法の結果ではない。そして、ステップ930においてエラーメッセージが生成される。ステップ930の実行後、またはもしステップ920において、計算された添字参照が正規の値であると判断された場合、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0212】ステップ820において、評価中の内部疑似コードのノードが「savptr」コマンドを有するかどうかを定めるテストが、「実行時エラー検査プロセス」によって行われる。上に述べたように、「savptr」コマンドが「翻訳された原始コード」に挿入されている。したがって、実行時において評価されると、各「savptr」コマンドによって、ポインタ検査表200内の適切な横列に含まれるポインタ情報がポインタ保存スタック65にコピーさせられる。

【0213】もしステップ820において、ノードが「savptr」コマンドを有すると判断された場合、ステップ825において評価器35が、コピーされているポインタのアドレスをインタブリタスタック60の最上部から検索により取り出す。その後、評価器35が、インタブリタスタック60から検索により取り出されたアドレスを用いてポインタ検査表200の適切な横列を探索し、ポインタ情報をポインタ検査表200の横列からポインタ保存スタック65にコピーする。

【0214】これに加えて、もし、上に述べたように、「savptr」コマンドが関数を指示するポインタを有する場合、この、関数を指示するポインタはポインタ保存スタック65にも入れられる。

【0215】上に述べたように、ポインタ情報は、図17に示し下に説明するような「実行時ポインタ検査表更新サブルーチン」のステップ1060からステップ1080までにおいてポインタ検査表200に入れるために、評価器35によってポインタ保存スタック65から次に検索により取り出される。ステップ825の実行後、手順はステップ800に戻り、上記のように進む。

【0216】実行時エラー検査プロセスによって評価中の内部疑似コードのノードが「tblupd」コマンドを有することがステップ830（図13）において検出された場合、手順はステップ1002において、図15から図17までに示す「実行時ポインタ検査表更新サブルーチン」に入る。

【0217】ステップ1002（図15）において、「tblupd」コマンドのノードに含まれる特徴情報が評価され、それからステップ1004からステップ1090（図17）までにおいて、型のマッチングが得られるまで、いくつかの条件に対して検査される。なお、

割り当てられるポインタのアドレスは、実行時においてインタプリタスタック60から得られる。割り当てられるポインタについてのポインタ情報で更新する必要があるポインタ検査表200内の適切な横列を探索特定するのにポインタのアドレスが用いられる。

【0218】ステップ1004において、識別特定された変数または関数のアドレスがポインタに割り当てられる内部記号表75内のエントリ、を指示するポインタがノードに含まれるかどうか、を定めるテストが行われる。

【0219】もしステップ1004において、内部記号表75内のエントリを指示するポインタがノードに含まれると判断された場合、そのポインタは、識別特定された変数または関数のアドレスおよび、もし利用可能なら、識別特定された変数または関数のサイズを検索により取り出すために、ステップ1006において内部記号表75の適切なエントリにアクセスするのに用いられる。

【0220】ステップ1008において、内部記号表75内でサイズ情報が利用可能だったかどうかを定めるテストが行われる。もしステップ1008において、内部記号表75内でサイズ情報が利用可能ではなかったと判断された場合、ステップ1010において、状態フラグが「不明」(UNKNOWN)にセットされる。

【0221】もしステップ1008において、内部記号表75内でサイズ情報が利用可能だったと判断された場合、ステップ1014において、そのポインタが関数または変数を指示しているかどうかを定めるテストが行われる。

【0222】もしステップ1014においてポインタが関数を指示していると判断された場合、ステップ1016において、状態フラグが「関数指示」(PFUNC)にセットされる。この場合、関数は始点からしか初期化されないで、関数を指示するポインタについての有効領域範囲はその関数の始点だけとなる。したがって、ステップ1017において有効領域範囲が、ステップ1006において内部記号表から検索により取り出された始点アドレスにセットされる。

【0223】もしステップ1014においてポインタが変数を指示していると判断された場合、ステップ1018において、状態フラグが「境界あり」にセットされる。ステップ1019において、ポインタについての有効領域範囲が、内部記号表75から検索により取り出されたアドレスおよびサイズ情報を用いて計算される。

【0224】ステップ1020において、ステップ1006からステップ1019までの実行時に割り当てられるポインタについて定義された情報が、割り当てられるポインタについて設定されたポインタ検査表200の適切な横列に追加される。その後、手順はステップ800(図13)において「実行時エラー検査プロセス」に戻

る。

【0225】もしステップ1004(図15)において、内部記号表75内のエントリを指示するポインタがノードに含まれないと判断された場合、手順はステップ1022に進む。

【0226】ステップ1022においては、ノードが「readintstk」コマンドだけを含むかどうかを定めるテストが行われる。このコマンドは、もしポインタが第2のポインタの内容を割り当てられる場合には、読み出し時においてパズ木に入れられる。

【0227】もしステップ1022において、ノードが「readintstk」コマンドだけを含むと判断された場合は、コピーされる第2のポインタのアドレスを検索により取り出すために、評価器35がインタプリタスタック60にアクセスするのが好ましい。その後、ステップ1026において、評価器35が、ステップ1024においてインタプリタスタック60から検索により取り出されたアドレスを用いて第2のポインタに対するポインタ検査表200内の横列を探索する。

【0228】それから、第2のポインタについてポインタ検査表200から検索により取り出された情報が、割り当てられるポインタに対する横列にコピーされる。その後、手順はステップ800(図13)において「実行時エラー検査プロセス」に戻る。

【0229】もしステップ1022(図15)において、ノードが「readintstk」コマンドを含まないと判断された場合、手順はステップ1030(図16)へ進む。

【0230】ステップ1030においては、ノードが、ポインタに割り当てられる文字列のアドレスおよびサイズを含むかどうかを定めるテストが行われる。もしステップ1030においてノードが、ポインタに割り当てられる文字列のアドレスおよびサイズを含むと判断された場合には、ステップ1032において評価器35が、アドレスおよびサイズ情報に基づいてポインタについての有効な領域範囲を計算する。

【0231】ステップ1034において状態フラグを「境界あり」にセットした後、割り当てられるポインタに対してステップ1032およびステップ1034の実行中に定義された情報が、ステップ1036において、割り当てられるポインタについてのポインタ検査表200の適切な横列に入れられる。その後、手順はステップ800(図13)において「実行時エラー検査プロセス」に戻る。

【0232】もしステップ1030においてノードが、ポインタに割り当てられる文字列のアドレスおよびサイズを含まないと判断された場合、手順はステップ1038に進む。

【0233】ステップ1038において、ポインタが不正な値を割り当てられことからノードが「不正」(IL

LEGAL) の状態表示を含むかどうかを定めるためのテストが行われる。

【0234】もしステップ1038において、ノードが「不正」の状態表示を含むと判断された場合、ステップ1040において状態フラグが「不正」にセットされる。そして、ポインタについて定義された情報がステップ1042において、ポインタ検査表200の適切な横列に追加される。その後、手順はステップ800 (図13) において「実行時エラー検査プロセス」に戻る。

【0235】もしステップ1038において、ノードが「不正」の状態表示を含まないと判断された場合、手順はステップ1044に進む。

【0236】ステップ1044においては、ノードが「readintstk」コマンドを構造体メンバのサイズとともに含むかどうかを定めるテストが行われる。もしステップ1044において、ノードが「readintstk」コマンドを構造体メンバのサイズとともに含むと判断された場合、ステップ1046において評価器35が、割り当てられる構造体メンバのアドレスを検索により取り出す。

【0237】なお、「readintstk」コマンドは、翻訳された原始コードに、読み出し時中に位置を占めており、その位置占めは、構造体メンバのアドレスがインタプリタスタック60の最上部にある時点にこのコマンドが評価されることが可能となるような仕方である。

【0238】ステップ1048において評価器35が、サイズ情報および検索により取り出されたアドレス情報に基づいて、ポインタについての有効領域範囲を計算し、それからステップ1050において、状態フラグを「境界あり」にセットする。ステップ1052において、割り当てられるポインタについて定義された情報がポインタ検査表200の適切な横列に追加される。その後、手順はステップ800 (図13) において「実行時エラー検査プロセス」に戻る。

【0239】もしステップ1044において、ノードが「readintstk」コマンドを構造体メンバのサイズとともに含まないと判断された場合、手順はステップ1060 (図17) に進む。

【0240】ステップ1060においては、ノードが「rdptrstk」コマンドを含むかどうかを定めるテストが行われる。もしステップ1060において、ノードが「rdptrstk」コマンドを含むと判断された場合、ステップ1064において評価器35が、ポインタ保存スタック65の内容を検索により取り出す。

【0241】上に述べたように、「rdptrstk」コマンドが検出されたときには、連関する「savptr」コマンドがステップ825 (図13) において既に評価されており、これによってポインタ検査表200を更新するために必要な適切なポインタ情報がポインタ保

存スタック65に入れられた。

【0242】ステップ1066においては、ポインタ保存スタックが、関数を指示するポインタを含むかどうかを定めるテストが行われる。このテストは、エラー検査タスクを翻訳された原始コードとコンパイルされたライブラリ関数とから構成されるソフトウェアプログラム上でも稼動可能にするために実現される。

【0243】なお、もしポインタ保存スタック65が、呼び出された関数へポインタ情報を送るために (ステップ545およびステップ547)、または呼び出された関数からポインタ情報を返すために (ステップ550およびステップ552) 用いられる場合には、関数を指示するポインタは、読み出し時において「savptr」コマンドに含まれる。

【0244】関数を指示するポインタは、呼び出された関数へポインタ情報を送る場合にはポインタ保存スタック65からポインタ情報を受け取るべき関数を識別特定し、または、呼び出し側関数へポインタ情報を返す場合にはポインタ保存スタック65にポインタ情報を入れた関数を識別特定する。

【0245】もしステップ1066において、ポインタ保存スタックが、関数を指示するポインタを含まないと判断された場合、ポインタ保存スタック65におけるポインタ情報の有効性をテストする必要はなく、手順は下に述べるステップ1080に進む。

【0246】もしステップ1066において、ポインタ保存スタックが、関数を指示するポインタを含むと判断された場合には、関数を指示するポインタの有効性をテストする必要がある。すなわちステップ1068において、ポインタ保存スタック65が、呼び出された関数へポインタ情報を送るために用いられているかまたは呼び出された関数からポインタ情報を返すために用いられているかを定めるためのテストが行われる。

【0247】もしステップ1068において、ポインタ保存スタック65が、呼び出された関数へポインタ情報を送るために用いられていると判断された場合、送られたポインタ情報が、呼び出された関数によって検索により取り出せるように呼び出し側関数によってポインタ保存スタック65に入れられたことを確認するために、ステップ1070において、検索により取り出された関数指示ポインタが、現に実行中の関数のアドレスに等しいかどうかを定めるテストが行われる。

【0248】もしステップ1070において、検索により取り出された関数指示ポインタが、現に実行中の関数のアドレスに等しくないと判断された場合、ポインタ保存スタック65からのポインタ情報は、ポインタ検査表200に入れるべきではない。そして手順はステップ800 (図13) において、「実行時エラー検査プロセス」に戻る。

【0249】もしステップ1070において、検索によ

り取り出された関数指示ポインタが、現に実行中の関数のアドレスに等しいと判断された場合、ポインタ保存スタック65からのポインタ情報は、ステップ1080においてポインタ検査表200に入れるべきである。その後、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0250】もしステップ1068において、ポインタ保存スタック65が、呼び出し側関数へポインタ情報を返すために用いられていると判断された場合、ポインタ情報が、呼び出し側関数によって検索して取り出せるように呼び出された関数によってポインタ保存スタック65に入れられたことを確認するために、ステップ1074において、検索により取り出された関数指示ポインタが、今実行を完了したばかりの関数のアドレス、すなわち呼び出された関数のアドレスに等しいかどうかを定めるテストが行われる。

【0251】もしステップ1074において、検索により取り出された関数指示ポインタが、今実行を完了したばかりの関数のアドレスに等しくないと判断された場合、ポインタ保存スタック65からのポインタ情報は、ポインタ検査表200に入れるべきではない。そして手順はステップ800（図13）において、「実行時エラー検査プロセス」に戻る。

【0252】もしステップ1074において、検索により取り出された関数指示ポインタが、今実行を完了したばかりの関数のアドレスに等しいと判断された場合、ポインタ保存スタック65からのポインタ情報は、ステップ1080においてポインタ検査表200に入れるべきである。その後、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0253】ステップ1080においてポインタ検査表200が、ステップ1064においてポインタ保存スタック65から検索により取り出されたポインタ情報を用いて更新される。その後、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0254】もしステップ1060において、ノードが「rdptrstk」コマンドを含まないと判断された場合、手順はステップ1090に進む。

【0255】ステップ1090においては、ノードが「不明」の状態表示を含むかどうかを定めるテストが行われる。もしステップ1090において、ノードが「不明」の状態表示を含むと判断された場合、ステップ1092において評価器35が、状態フラグを「不明」にセットし、それから、割り当てられるポインタについて定義された情報がステップ1094において、ポインタ検査表200の適切な横列に追加される。

【0256】その後、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0257】もしステップ1090（図17）において、ノードが「不明」の状態表示を含まないと判断され

た場合、手順はステップ1096に進む。

【0258】もし手順がステップ1096に到達した場合、「tblupd」ノードは、上に設定されたテストのうちのどれに基づく特徴付けもできないことになる。したがって、ステップ1096においてエラーメッセージが生成され、その後、手順はステップ1098において「実行時ポインタ検査表更新サブルーチン」から出て、サブルーチンが終了する。

【0259】実行時エラー検査プロセスによって評価中の内部疑似コードのノードが「ptrchk」コマンドを有することがステップ840（図13）において検出された場合、手順はステップ1104において、図18および図19に示す「実行時ポインタ検査サブルーチン」に入る。上に述べたように、「実行時ポインタ検査サブルーチン」は、関連解除される各ポインタの有効性のテストを行う。

【0260】ステップ1104において、検査されるポインタのアドレスを検索により取り出すために評価器35が、インタプリタスタック60にアクセスする。上に述べたように、「ptrchk」コマンドは、実行時において評価されているとき、検査されているポインタのアドレスがインタプリタスタック60の最上部に来るような仕方、パーズ木の中に位置させる。

【0261】ステップ1108において評価器35が、ステップ1104においてインタプリタスタック60から検索により取り出されたポインタアドレスを利用して、検査されているポインタに対するポインタ検査表200内の横列を探索する。

【0262】ステップ1112において、検査されているポインタに対するポインタ検査表200内の状態エントリ260が「PFUNF」にセットされているかどうかを定めるためのテストが行われる。もしステップ1112において、状態エントリ260が「関数指示」にセットされていると判断された場合、ステップ1116において関数のアドレスが内部記号表75から検索により取り出される。

【0263】ステップ1120においてポインタ検査表200内のポインタ内容エントリ240が、内部記号表75から検索により取り出されたアドレスによって書き直される。このルーチンは、翻訳器15にロードされた最新版の関数が実行されることを確実にするために実現される。その後、手順はステップ1124に進む。

【0264】ステップ1124において、ポインタ検査表200内のポインタ内容エントリ240に記録されている値が、ポインタの実際の内容に等しいかどうかを定めるためのテストが行われるのが好ましい。この好ましい実施例においては、評価器35が知らずに、「コンパイルされたコード」によって行われたポインタ修正を検出するために、上記ラップ関数実現例に追補されるメカニズムが設けられる。評価器35が前の修正を知らな

ったので、ポインタ検査表200は適切な更新が行われなかった。

【0265】もしステップ1124において、これらの値が等しくないと判断された場合、このポインタについて前に検出されなかった修正が発生していることになる。この修正は「コンパイルされたコード」によって有効正当に行われたものと仮定する。

【0266】そして、ステップ1128において、検査されるポインタについて状態フラグが「不明」にセットされ、それ以上の追加検査は行われない。このようにして、「コンパイルされたコード」による有効なポインタ修正についての誤った警告が防止される。その後、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0267】もしステップ1124において、記録されている内容がポインタの実際の内容に等しいと判断された場合、手順はステップ1132に進む。

【0268】ステップ1132において、検査されるポインタについてポインタ検査表200内の状態エントリ260が「不正」とセットされているかどうかを定めるためのテストが行われる。もしステップ1132において、状態が「不正」とセットされていると判断された場合、一定ゼロ値または負数のポインタが関連解除されるというエラーが今発生していることになる。その結果、手順は、図22を参照して下に述べる診断サブルーチンに進む。

【0269】もしステップ1132において、状態が「不正」とセットされていないと判断された場合、手順はステップ1140（図19）に進む。

【0270】ステップ1140（図19）においては、検査されるポインタについてポインタ検査表200内の状態エントリ260が「解放」とセットされているかどうかを定めるためのテストが行われる。

【0271】もしステップ1140において、状態が「解放」にセットされていると判断された場合には、割付を解除されていたメモリがアクセスされているという不適切な状態、すなわち解放メモリアクセスエラーが発生していることになる。その結果、手順は、図22を参照して下に述べる診断サブルーチンに進む。

【0272】もしステップ1140において、状態が「解放」にセットされていないと判断された場合には、手順はステップ1148に進む。

【0273】ステップ1148において、検査されるポインタについてポインタ検査表200内の状態エントリ260が「境界あり」とセットされているかどうかを定めるためのテストが行われる。

【0274】もしステップ1148において、状態が「境界あり」にセットされていると判断された場合には、ステップ1152において、検査されているポインタの実際の内容が、検査されているポインタについての

ポインタ検査表200の有効メモリ境界エントリ250a、250bに定められるような有効領域範囲内にあるかどうかを定めるためのテストが行われる。

【0275】もしステップ1152において、検査されているポインタの、検索により取り出されたポインタ値が、有効領域範囲内にあると判断された場合、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0276】しかし、もしステップ1152において、検索により取り出されたポインタ値が有効領域範囲内ないと判断された場合には、メモリアクセスエラーが発生している。その結果、手順は、図22を参照して下に述べる診断サブルーチンに進む。

【0277】もしステップ1148において、状態が「境界あり」にセットされていないと判断された場合には、手順はステップ1160に進む。

【0278】ステップ1160においては、検査されるポインタについてポインタ検査表200内の状態エントリ260が「割付済」にセットされているかどうかを定めるためのテストが行われる。

【0279】もしステップ1160において、状態が「割付済」にセットされていると判断された場合には、ステップ1162において、ポインタの内容が、図6に示す連関するメモリ割付構造体280の下側および上側境界エントリ282、284によって定義されるような割付されたメモリの有効な領域範囲の中にあるかどうかを定めるためのテストが行われる。

【0280】もしステップ1162において、ポインタの内容が、割付されたメモリについての有効な領域範囲の中にないと判断された場合には、メモリアクセスエラーが発生している。その結果、手順は、図22を参照して下に述べる診断サブルーチンに進む。

【0281】もしステップ1162において、ポインタの内容が、割付されたメモリについての有効な領域範囲の中にあると判断された場合には、ステップ1164において、ポインタが、割付されたメモリに対して読み出しまたは書き込みを行うために関連解除されるかどうかを定めるためのテストが行われる。

【0282】もしステップ1164において、ポインタが、割付されたメモリに対して書き込みを行うために関連解除されると判断された場合、上に述べた初期化ビットベクトル288の適切なビットが、ステップ1168において、割付されたメモリのうちの対応するバイトの新たな初期化状態を表示するように更新される。その後、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0283】もしステップ1164において、ポインタが、割付されたメモリに対して読み出しを行うために関連解除されると判断された場合、ステップ1172において、割付されたメモリのうちの対応するバイトが初期

化されているかどうかを定めるために、初期化ビットベクトル288の適切なビットがアクセスされる。

【0284】ステップ1174において、読み出しが行われる割付されたバイトが初期化されているかどうかを定めるためのテストが行われる。もしステップ1174において、これらのバイトが初期化されていないと判断された場合には、メモリアクセスエラーが発生している。その結果、手順は、図22を参照して下に述べる診断サブルーチンに進む。

【0285】もしステップ1174において、これらのバイトが初期化されていると判断された場合には、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0286】もしステップ1160において、状態が「割付済」にセットされていないと判断された場合には、手順はステップ1176に進む。もし手順がステップ1176に到達した場合、状態エントリ260（図2）に記録されている情報は、上に設定されたテストのうちのどれに基づく特徴付けもできないことになる。したがって、ステップ1176においてエラーメッセージが生成され、その後、手順は「実行時ポインタ検査サブルーチン」から出て、サブルーチンが終了する。

【0287】実行時エラー検査プロセスによって評価中の内部疑似コードのノードが、「コンパイルされた関数」についての「call」（呼び出し）コマンドを有することがステップ850（図13）において検出された場合、手順はステップ1202において、図20に示す「コンパイルされた関数」エラー検査プロセスに入る。

【0288】もし呼び出された「コンパイルされた関数」が下にさらに述べるように割付されたメモリ領域へのアクセスを有する場合に、「コンパイルされた関数」エラー検査プロセスによって、与えられた「コンパイルされた関数」に関連する「ラップ関数」を起動するためおよび図21に示す初期化ビットベクトル維持サブルーチンを起動するためのメカニズムが得られる。

【0289】ステップ1202において、呼び出された「コンパイルされた関数」が、連関する「実行前ラップ関数」を有するかどうかを定めるためのテストが、「コンパイルされた関数」エラー検査プロセスによって行われる。

【0290】もしステップ1202において、呼び出された「コンパイルされた関数」が、連関する「実行前ラップ関数」を有すると判断された場合、上に述べたような、必要なポインタ検査およびその他の関数を行うために、ステップ1204において、「実行前ラップ関数」が実行される。「実行前ラップ関数」は、連関する「コンパイルされた目的コード関数」と同じ引数を送られる。その後、手順はステップ1205に進む。

【0291】もしステップ1202において、呼び出さ

れた「コンパイルされた関数」が、連関する「実行前ラップ関数」を有しないと判断された場合、手順はステップ1205に進む。

【0292】ステップ1205においては、割付アクセスフラグが、呼び出された「コンパイルされた関数」に対応する内部記号表75内のエントリにセットされているかどうかを定めるためのテストが行われる。割付アクセスフラグは、もしこの関数が割付されたメモリのどれかにアクセスを有する場合にステップ337において読み出し時にセットされた。

【0293】もしステップ1205において、割付アクセスフラグがセットされていると判断された場合、ステップ1207において、図21に示す「初期化ビットベクトル維持サブルーチン」が実行される。手順がステップ1210においてこのサブルーチンに入ることによって、翻訳器15によっては通常に検出されない割付されたメモリの初期化の「コンパイルされた関数」による検出のためのメカニズムが得られる。

【0294】なお、割付されたメモリの「翻訳された関数」による初期化は、図19に示す「実行時ポインタ検査サブルーチン」のステップ1164およびステップ1168の実行時に検出される。このようにして、初期化ビットベクトル288は、初期化が「コンパイルされた関数」または「翻訳された関数」のいずれによって行われたかに関係なく、割付されたメモリ領域の各対応するバイトの初期化状態を適切に記録する。

【0295】上に述べたように、本発明の好ましい実施例においては、メモリ初期化の検出を容易にするために、初期化されていない自動変数および割付されたメモリ領域がすべて、例えば「FFFA 5A5A」のような既知の4バイトの16進語によって事前マーク付けされる。

【0296】ステップ1210（図21）において「コンパイルされた関数」が呼び出された時点で、初期化されていない対応するバイトのすべてを識別特定するために、評価器35が、「コンパイルされた関数」によってアクセスされる割付されたバイトに対応する初期化ビットベクトル288（図6）の各ビットの解析を行う。

【0297】それから、「コンパイルされたコード」によってアクセスされる初期化されていない対応するバイトのすべてを識別特定するために、ステップ1220において評価器35が、巡回冗長テスト（CRCテスト）を行う。ステップ1225において、「コンパイルされた関数」が適切な引数で呼び出され、実行される。

【0298】「コンパイルされたコード」の実行に続いて、手順はステップ1230に進み、ここで、「コンパイルされたコード」の実行前には「コンパイルされたコード」がアクセスでき初期化されていなかったバイトのすべてに対して第2のCRCテストが行われる。

51

【0299】次にステップ1235において、CRCテストによってバイトの値が変化したものがあるかどうか、ステップ1220において行われたCRCテストの結果が、ステップ1230において行われたCRCテストの結果に等しいかどうか、を定めるためのテストが行われる。もしステップ1235において、これら2つのCRCテスト値が等しくないと判断された場合には、手順はステップ1240に進む。

【0300】ステップ1240においては、割付されたメモリブロック全体を「初期化済」とマーク付けすべきかどうか、または初期化されたブロック内のバイトだけを「初期化済」とマーク付けすべきかについて定めるために、ユーザによって定義されたフラグ（ユーザ定義フラグ）が解析される。ユーザ定義フラグは、各ソフトウェアデバッグ処理プロセスの開始時にユーザによってセットされるのが好ましい。

【0301】このユーザ定義フラグによって、ユーザは、「コンパイルされた関数」がメモリを既知の事前マークされた16進値で初期化する結果、すなわちメモリを16進値「FFFA 5A5A」で初期化する結果から来る誤ったエラー発生警告を防止することが可能となる。

【0302】次にステップ1245において、割付されたメモリブロック全体を、初期化状態（ただ1個のバイトしか「コンパイルされたコード」によって初期化されなかったような初期化状態も含む）としてマーク付けすべきことを表すようにユーザ定義フラグがセットされているかどうか、または代わりに、初期化されていると判断されたバイトだけを初期化状態としてマーク付けすべきであると指定されているかどうか、を定めるためのテストが行われる。

【0303】もしステップ1245において、ユーザが、割付されたメモリブロック全体を初期化状態としてマーク付けすべきことを指定している、と判断された場合には、ステップ1250において、初期化ビットベクトル288内の各ビットが、初期化状態を表すように更新される。

【0304】もしステップ1245において、ユーザが、実際に初期化されているバイトだけを初期化状態としてマーク付けすべきことを指定している、と判断された場合には、手順はステップ1255に進む。

【0305】このステップ1255においては、割付されたメモリ領域において「FFFA5A5A」以外の内容を有する4バイトブロックの各々、すなわちもはや既知の事前マーク値を持たないバイト、が識別特定される。その後、ステップ1258において、初期化ビットベクトル288内の対応するビットが、初期化状態を表すように更新される。

【0306】代わりの実施例においては、「コンパイルされた関数」がアクセスする初期化されていないメモリ

52

のすべてのバイトが、ステップ1225における「コンパイルされたコード」の実行に先立ってセーブ（保存）される。

【0307】この場合、もしステップ1235において、「コンパイルされたコード」が割付されたメモリを初期化していると判断されたならば、ステップ1255において、初期化されておらず且つアクセス可能なバイトの内容を「コンパイルされた目的コード」の実行に先立って記憶セーブされた値と比較することによって、初期化されたバイトが識別特定される。この実施例においては、割付されたメモリ領域を既知の値、すなわち16進値「FFFA 5A5A」で事前マーク付けする必要はない。

【0308】ステップ1250またはステップ1258の実行後、またはもしステップ1235において、「コンパイルされたコード」が、割付されたメモリのどのバイトも初期化していないと判断された場合、手順はステップ1265（図20）において「「コンパイルされた関数」エラー検査プロセス」に戻る。

【0309】もしステップ1205（図20）において、割付アクセスフラグがセットされていないと判断された場合、呼び出された「コンパイルされた関数」は、どの割付されたメモリにもアクセスできない。また「初期化ビットベクトル維持サブルーチン」を実行する必要がない。

【0310】したがって、もしフラグがセットされていない場合には、「コンパイルされた目的コード」が適切な引数で呼び出され、実行される。「コンパイルされた目的コード」が実行された後、手順はステップ1265（図20）において「「コンパイルされた関数」エラー検査プロセス」に戻る。

【0311】ステップ1265においては、呼び出された「コンパイルされた関数」が連関する「実行後ラップ関数」を有するかどうかを定めるためのテストが、「「コンパイルされた関数」エラー検査プロセス」によって行われる。

【0312】もしステップ1265において、呼び出された「コンパイルされた関数」が連関する「実行後ラップ関数」を有すると判断された場合、ステップ1270において、「実行後ラップ関数」が、その連関する「コンパイルされた目的コード関数」によって返された値で実行される。

【0313】「実行後ラップ関数」は、返されたまたは形成されたポインタについて必要なポインタ検査表更新タスク、および上に述べたようなその他必要なタスクを行う。その後、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0314】もし図18および図19に示す「実行時ポインタ検査サブルーチン」の実行中にエラーが検出された場合、すなわちステップ1132、1140、115

2、1162、または1174のテスト条件が不成功だった場合、手順は、図22に示す診断サブルーチンに進む。

【0315】手順がステップ1310において「診断サブルーチン」に入ると、エラーメッセージが生成される。このエラーメッセージは、診断情報、すなわちメモリアクセスエラーの種類、およびポインタ検査表200のファイル/ライン番号エントリ270から取り出された、ポインタに最後に修正が行われたライン番号、からなる。

【0316】その後、プログラマは、検出されたエラーを訂正するためにステップ1320において、上に述べたように読み出し期間の手段にアクセスできる。プログラマによるエラー訂正が終ると、手順はステップ800（図13）において「実行時エラー検査プロセス」に戻る。

【0317】[メモリ漏洩の特定および検出されていないポインタ修正のソースの特定] 本発明の別の特徴によれば、メモリ漏洩、すなわち割付されたがもはやアクセス不可能なメモリスペースが、自動的に、またはユーザがメモリ漏洩検出アルゴリズムを起動することによって、検出される。

【0318】メモリ漏洩は一般に、割付されたメモリの第1のブロックを指示するポインタが、第1のブロックについての割付解除なしに、割付されたメモリの第2のブロックを指示するように再度割り当てられるときに生じる。周知のように、メモリ漏洩は全体の動作性能の漸増的な劣化をもたらす。

【0319】上に述べたように、「メモリ割付関数」によるメモリブロックの割付がされると、連関するメモリ割付構造体280（図6）がその連関する「実行後ラップ関数」によって形成されることが好ましい。メモリ割付構造体280は、上に述べたような、割付されたメモリに関するある情報（例えば割付されたメモリのの中の連関するブロックを現に指示するすべてのポインタのチェインリスト部分286内のリスト）を記録する。

【0320】上に述べたように、新たなポインタが、割付されたメモリを指示するように割り当てられる都度、その新たなポインタはチェインリストに追加される。同様に、割付されたメモリの第1のブロックを前に指示したポインタが、割付されたメモリの第2のブロックを指示するように再割当される都度、第2のブロックに連関するチェインリストに追加される前に、第1のブロックに連関するチェインリストから除去されることが好ましい。

【0321】したがって、メモリ割付構造体280のチェインリスト部分286に記録されている情報は、割り付けられたメモリのうちの連関するブロックを現に指示するポインタだけを表示する。そして、もしチェインリスト部分286の内容が空である場合には、割り付けら

れたメモリを指示するポインタで、割り付けられたメモリにアクセスするために用いられるようなポインタは存在しない。すなわち、メモリ漏洩が発生しているわけで、エラーメッセージを生成する必要がある。

【0322】上に述べたように、ポインタが、「コンパイルされたコード」によって、評価器35に知られずに、修正される場合を検出するための、ステップ1124およびステップ1128における追補のメカニズムが、図18および図19に示す「実行時ポインタ検査サブルーチン」によって得られる。

【0323】評価器35が、「コンパイルされたコード」によって行われた前の修正を知らなかったため、ポインタ検査表200はこの修正の時点で適切な更新が行われなかった。通補のメカニズムは、検査されるポインタについての状態フラグを「不明」状態にセットし、それ以上のポインタ追加検査は行わない。

【0324】このルーチンは「コンパイルされたコード」による有効なポインタ修正についての誤った警告を防止はするが、修正値がポインタ検査表200に入れられるように、「コンパイルされたコード」による各修正を検出する方が好ましい。

【0325】したがって、「不明」の状態表示を有するすべてのポインタを識別特定するために、ポインタ検査表200の各横列の状態エントリ260をユーザがサーチできるようなメカニズムが好ましい。

【0326】「不明」の表示を有する各ポインタについて、それぞれのポインタを識別特定する診断メッセージが、ファイル/ライン番号エントリ270からの情報とともに生成される。ライン番号情報は、ポインタが最後に更新された時点を表示する。このことから、プログラマは診断情報を用いて、翻訳器が知らなくても、ポインタ値を修正した「コンパイルされたコード」を特定することが可能となる。

【0327】その後、上に述べたような方法で、「コンパイルされたコード」に対する「実行後ラップ関数」が書かれ、これによって、「不明」状態がさらに発生するのを防止するために、「コンパイルされた関数」の実行に続いて新たなポインタ情報がポインタ検査表200に追加される。

【0328】プログラム開発中にプログラマは、CINリセットコマンドのようなリセットコマンドを起動することによって翻訳器15を頻繁にリセットすることを望んでいる。周知のように、このコマンドは翻訳器15に、読み出しプロセスの終了時に元あった状態と同じ状態に戻るよう命令する。實際上、このことは、すべてのデータがその当初の値に再初期化され、「BSS」データが「0」に割り当てられる結果となる。

【0329】ポインタ検査表200は、CINリセットコマンド実行終了時に、読み出しプロセスの終了時にポインタ検査表200があった状態と同じ状態に戻るこ

が好ましい。したがって、ポインタ検査表200を再初期化し、それから、上に述べたように、すなわち他の静的データを指示するように初期化された静的ポインタおよび主関数に送られた「argv」ポインタ引数について、内部疑似コードの実行に先立ってポインタ検査表200にプレロードされた初期化条件を再ロードする必要がある。

【0330】以上の説明は、本発明の一実施例に関するもので、この技術分野の当業者であれば、本発明の種々の変形例を考え得るが、それらはいずれも本発明の技術的範囲に包含される。

【0331】

【発明の効果】以上述べたごとく、本発明によれば、「翻訳された原始コード」と「コンパイルされた目的コード」との両方を実行しながらエラー検出タスクを行うことのできるソフトウェアテストおよびデバッグ処理ツールが得られる。また、本発明によれば、与えられたポインタによって指示されたメモリがそれぞれのポインタについての適切な境界範囲内にあることを確実にするソフトウェアテストおよびデバッグ処理が可能となる。

【0332】さらに、読み出し時またはパース時に導出される情報を利用することにより、実行時に行われる互いに重複するポインタ検査の数を減少させて、より効率的なソフトウェアテストおよびデバッグ処理を行うことができる。

【図面の簡単な説明】

【図1】本発明に基づくメモリアクセスエラー検出システムを示す概略ブロック図である。

【図2】図3の原始コードファイルの例に宣言されている各ポインタについての現ポインタ情報を維持するポインタ検査表である。

【図3】図2のポインタ検査表に列記されるポインタを宣言し初期化する原始コードファイルの例を示す説明図である。

【図4】図3の原始コードファイルのライン10からライン60までの実行後の、いくつかのポインタと、それらのポインタによって指示されるそれぞれのメモリスペースとの間の関係を示す説明図である。

【図5】図3の原始コードファイルのライン70からライン100までの実行後の、いくつかのポインタと、それらのポインタによって指示されるそれぞれのメモリスペースとの間の関係を示す説明図である。

【図6】図3の原始コードファイルのライン200からライン210までの実行後の、ポインタ「ptr_all loc」と、このポインタによって指示される割付されたメモリブロックとの間の関係、およびこの割付されたメモリブロックについての追加情報を維持する連関メモリ割付構造体を示す説明図である。

【図7】パース木（構文木）を解析する際にメモリエラー検出コード挿入器によって用いられるような読み出し

時エラー検査プロセスの例を示す流れ図である。

【図8】図7のエラー検査プロセスによって用いられるような読み出し時配列次元検査サブルーチンの例を示す流れ図である。

【図9】図7のエラー検査プロセスによって用いられるような読み出し時ポインタ表更新サブルーチンの例を、図10と一体で示す流れ図である。

【図10】図7のエラー検査プロセスによって用いられるような読み出し時ポインタ表更新サブルーチンの例を、図9と一体で示す流れ図である。

【図11】図7のエラー検査プロセスによって用いられるような読み出し時ポインタ検査サブルーチンの例を、図9と一体で示す流れ図である。

【図12】コンパイルされたメモリ割付解除関数を実行しながら評価器が用いるようなメモリ割付解除モニタ処理サブルーチンの例を示す流れ図である。

【図13】実行時において内部疑似コードのノードを解析しながら評価器によって用いられるような実行時エラー検査プロセスの例を示す流れ図である。

【図14】図13のエラー検査プロセスによって用いられるような実行時配列次元検査サブルーチンの例を示す流れ図である。

【図15】図13のエラー検査プロセスによって用いられるような実行時ポインタ表更新サブルーチンの例を、図16および図17と一体で示す流れ図である。

【図16】図13のエラー検査プロセスによって用いられるような実行時ポインタ表更新サブルーチンの例を、図15および図17と一体で示す流れ図である。

【図17】図13のエラー検査プロセスによって用いられるような実行時ポインタ表更新サブルーチンの例を、図15および図16と一体で示す流れ図である。

【図18】図13のエラー検査プロセスによって用いられるような実行時ポインタ検査サブルーチンの例を、図19と一体で示す流れ図である。

【図19】図13のエラー検査プロセスによって用いられるような実行時ポインタ検査サブルーチンの例を、図18と一体で示す流れ図である。

【図20】図13のエラー検査プロセスによって用いられるような「コンパイルされた関数」エラー検査プロセスの例を示す流れ図である。

【図21】図20の「コンパイルされた関数」エラー検査プロセスによって用いられるような初期化ビットベクトル維持サブルーチンの例を示す流れ図である。

【図22】図18および図19の実行時ポインタ検査サブルーチンによって用いられるような診断サブルーチンの例を示す流れ図である。

【図23】ポインタ「ptr_part」を宣言し初期化する原始コードファイルの例を示す説明図である。

【図24】図23に示す原始コードファイルのデータ流れ解析中に設定される流れセット例を示す説明図であ

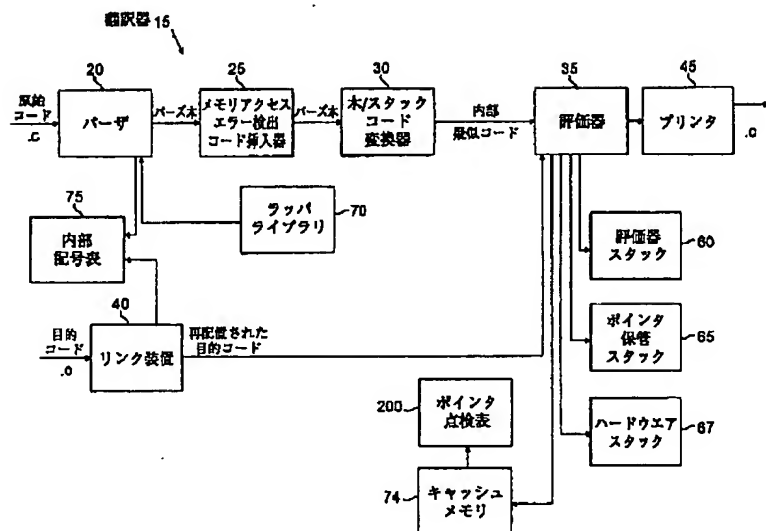
る。

【符号の説明】

15 翻訳器 (インタプリタ)
 20 パーザ (構文解析器)
 25 メモリアクセスエラー検出コード挿入器
 30 木/スタックコード変換器
 35 評価器
 40 リンク装置
 45 プリンタ
 60 インタプリタスタック
 65 ポインタ保存スタック
 67 ハードウェアスタック
 70 ラッパライブラリ
 74 キャッシュメモリ

75 内部記号表
 200 ポインタ検査表
 220、222、224 横列
 230 ポインタアドレスエントリ
 240 ポインタ内容エントリ
 250 a、250 b 有効メモリ境界エントリ
 260 状態エントリ
 270 ファイル/ライン番号エントリ
 280 メモリ割付構造体
 282 下側境界部分
 284 上側境界部分
 286 チェインリスト部分
 288 初期化ビットベクトル
 290 状態部分

【図1】



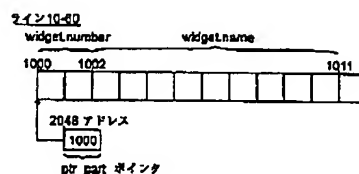
【図2】

ポインタ点検表 200

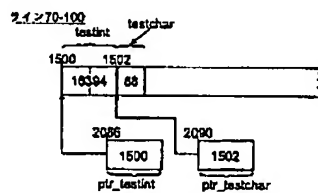
横列	ポインタ	ポインタ アドレス	ポインタ 内容	有効メモリ 境界		状態	ファイル/ライン 番号
				下側	上側		
220	ptr_part	2048	1000	1000	1011	境界あり	テスト ファイル 160
222	ptr_testint	2088	1500	1500	1501	境界あり	テスト ファイル 160
	...						
	ptr_alloc	2200	3000	3500		割付済	テスト ファイル 1210
	...						
224	ptr_testchar	2090	1502	1502	1502	境界あり	テスト ファイル 1100

230 エントリ 240 エントリ 250a エントリ 250b エントリ 260 エントリ 270 エントリ

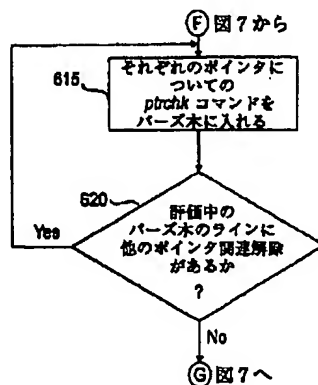
【図4】



【図5】



【図11】



【図3】

原始コードファイル=「テストファイル」

```

110 struct part {
120     int number;
130     char name[10];
140 } widget;
150 struct part *ptr_part;
160 ptr_part = &widget;

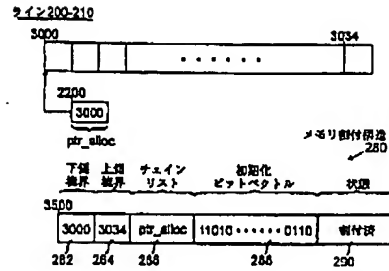
170 int testInt = 16394;
180 char testchar = 88;
190 int *ptr_testint = &testInt;
1100 char *ptr_testchar = &testchar;

1200 char *ptr_alloc;
1210 ptr_alloc = malloc(35 * sizeof(char));

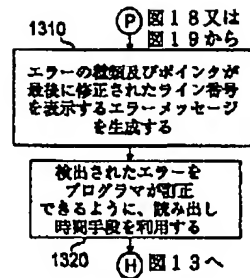
1300 ptr_testchar = &widget.name[0];
1310 *ptr_testint = 12024;

```

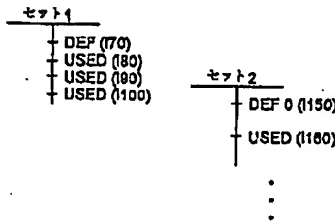
【図6】



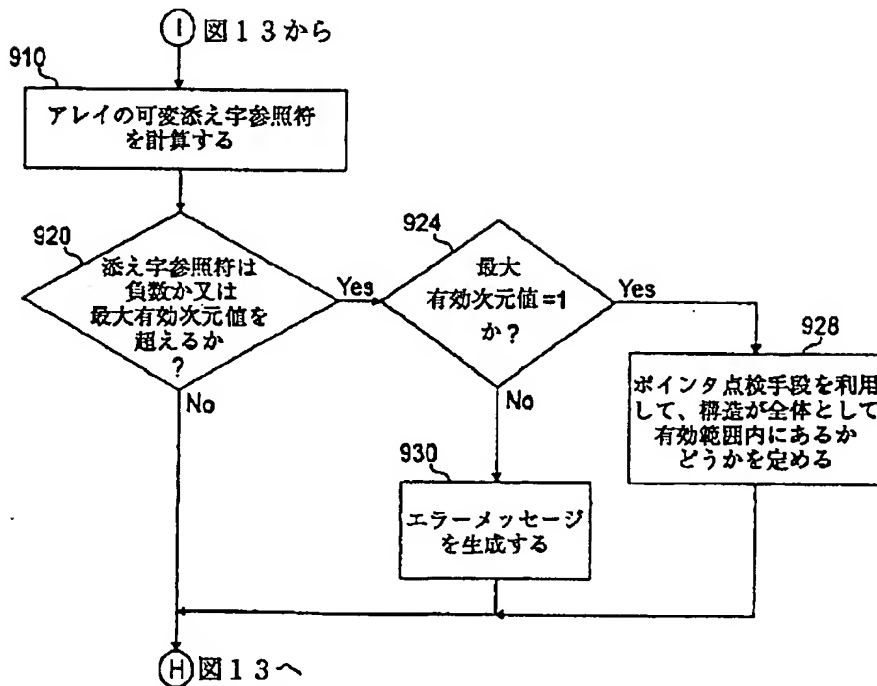
【図22】



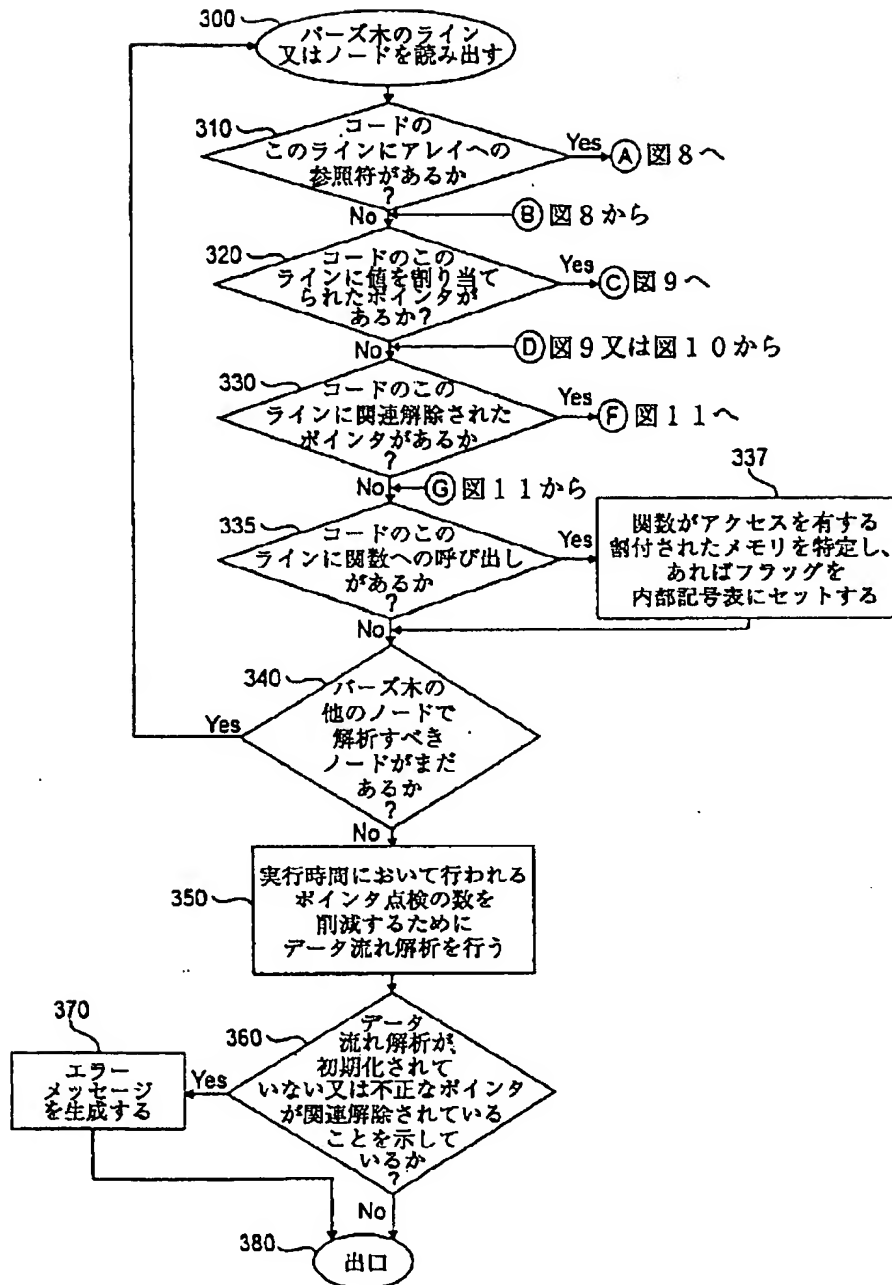
【図24】



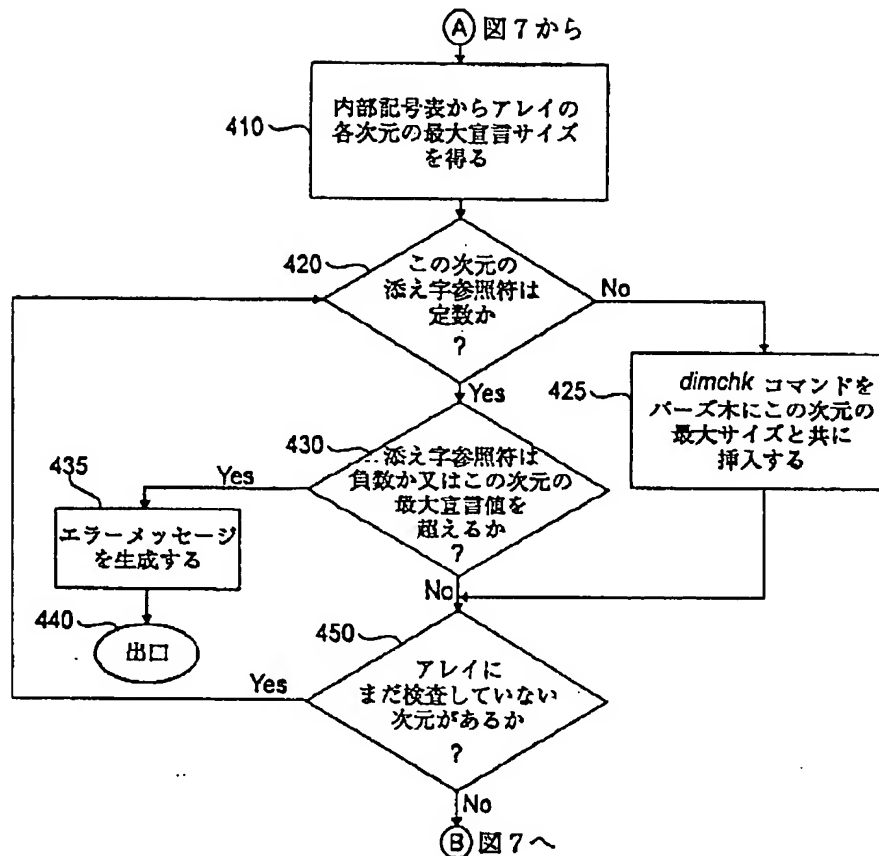
【図14】



【図7】



【図8】



【図23】

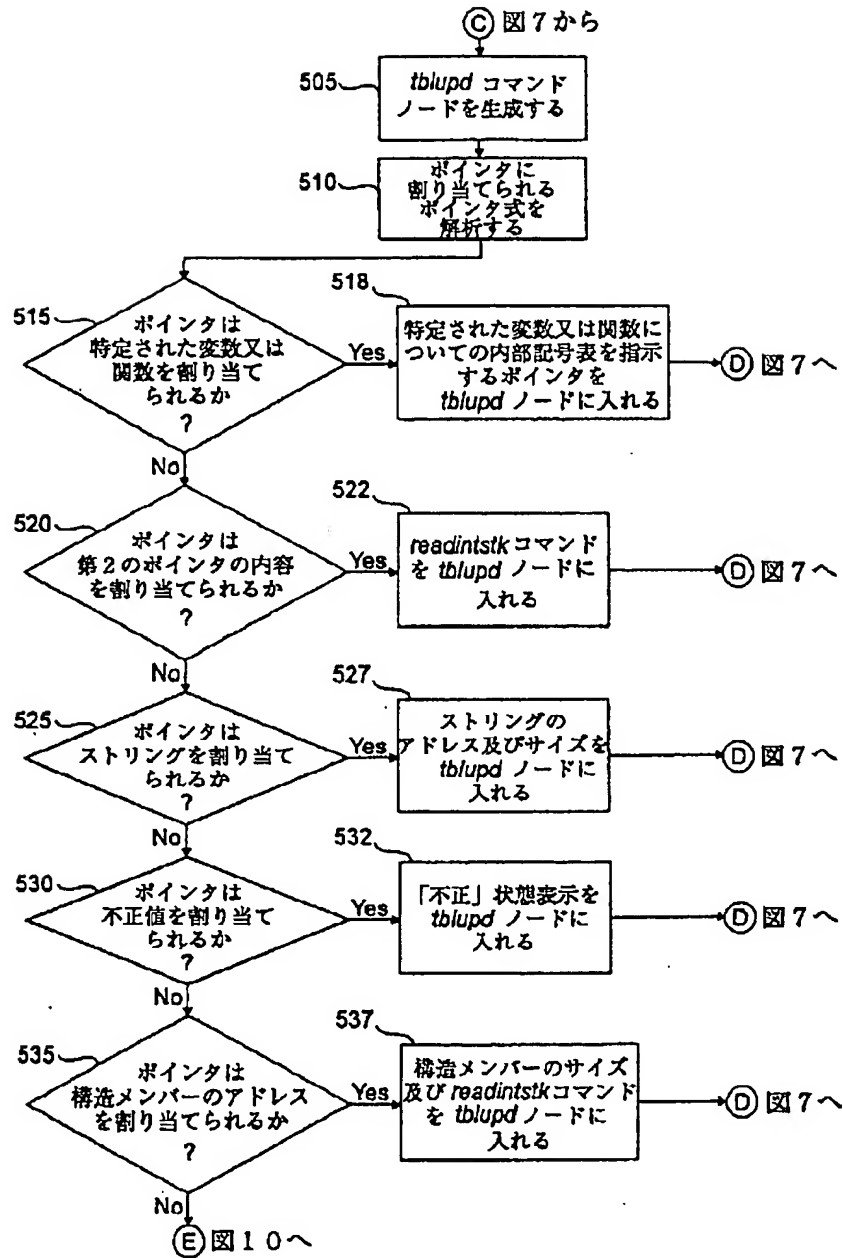
原始コードファイル「サンプル」

```

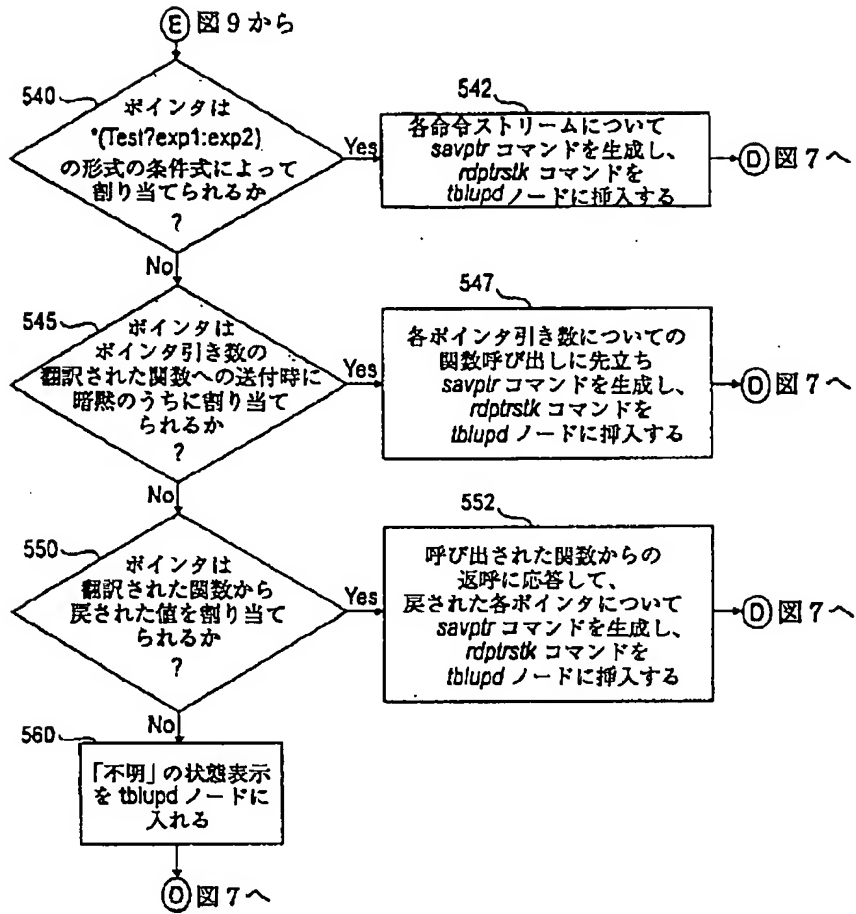
110 struct part {
120     int number;
130     char name [10];
140     char color;
150 } widget;
160 struct part *ptr_part;
170 ptr_part = &widget;
180 ptr_part->number = 880;
190 strcpy (ptr_part->name, "chair");
1100 ptr_part->color = 'b';
.
.
1150 ptr_part = 0;
1180 ptr_part->number = 870;
.
.

```

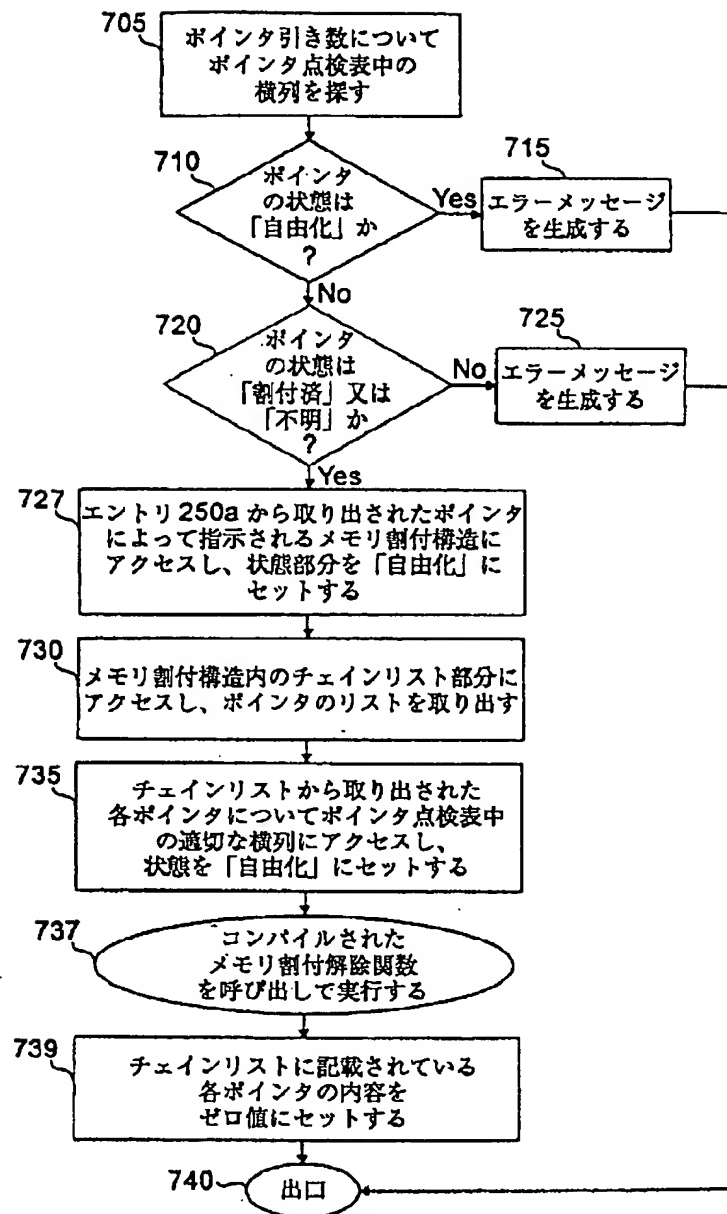
【図9】



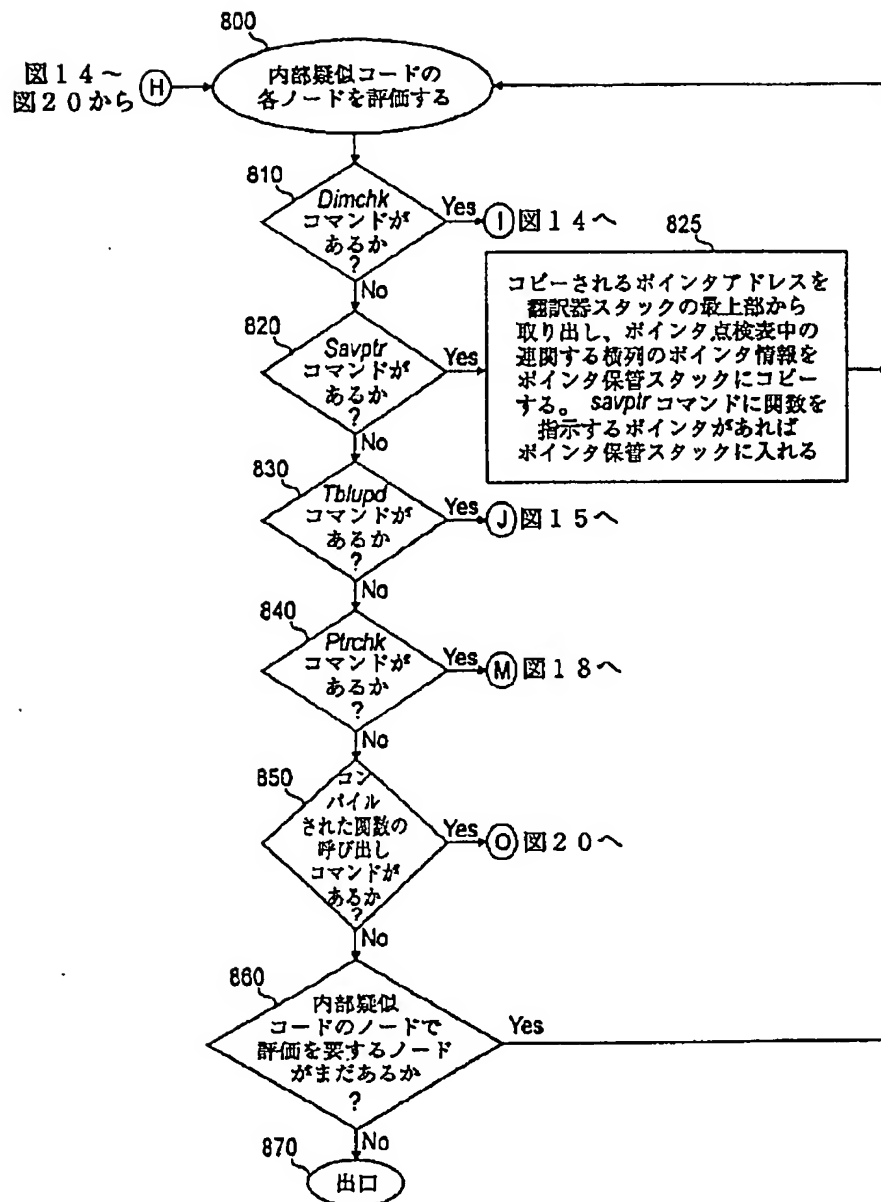
【図10】



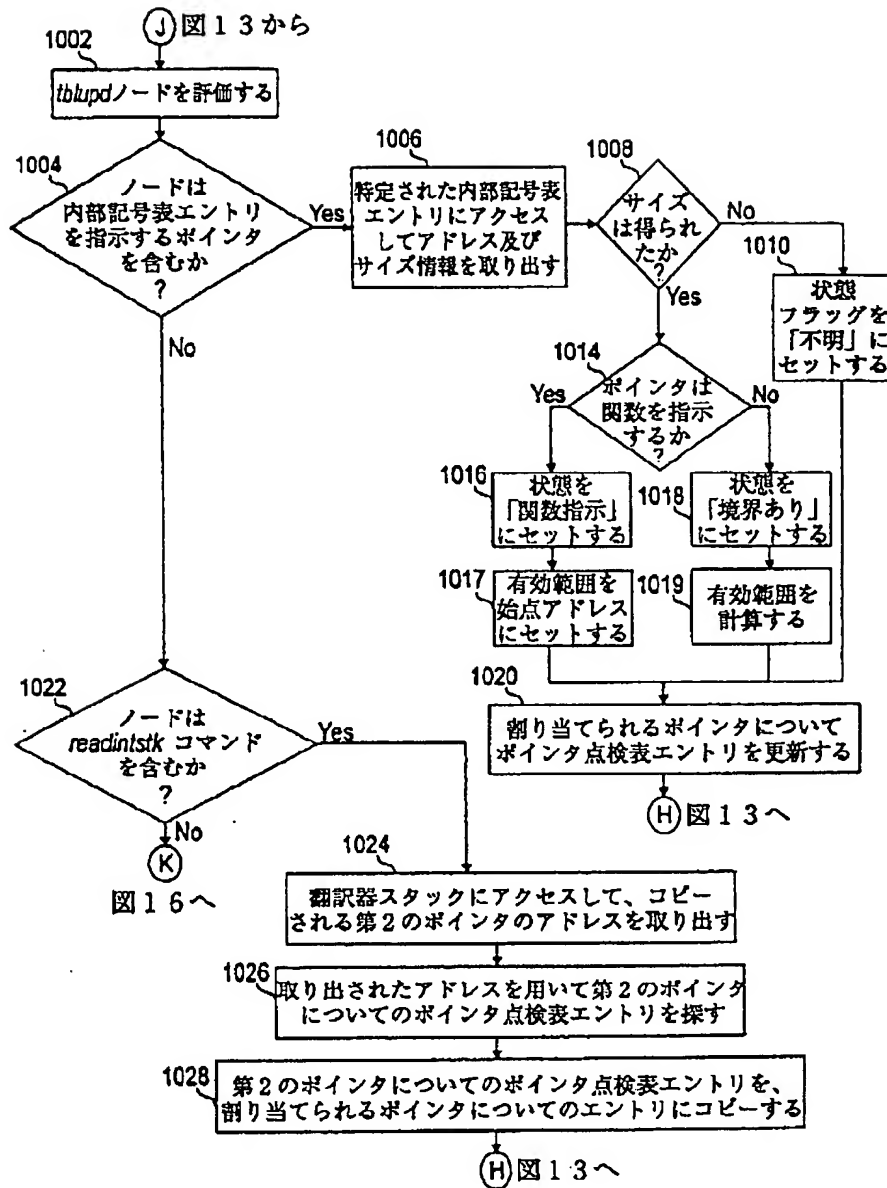
【図12】



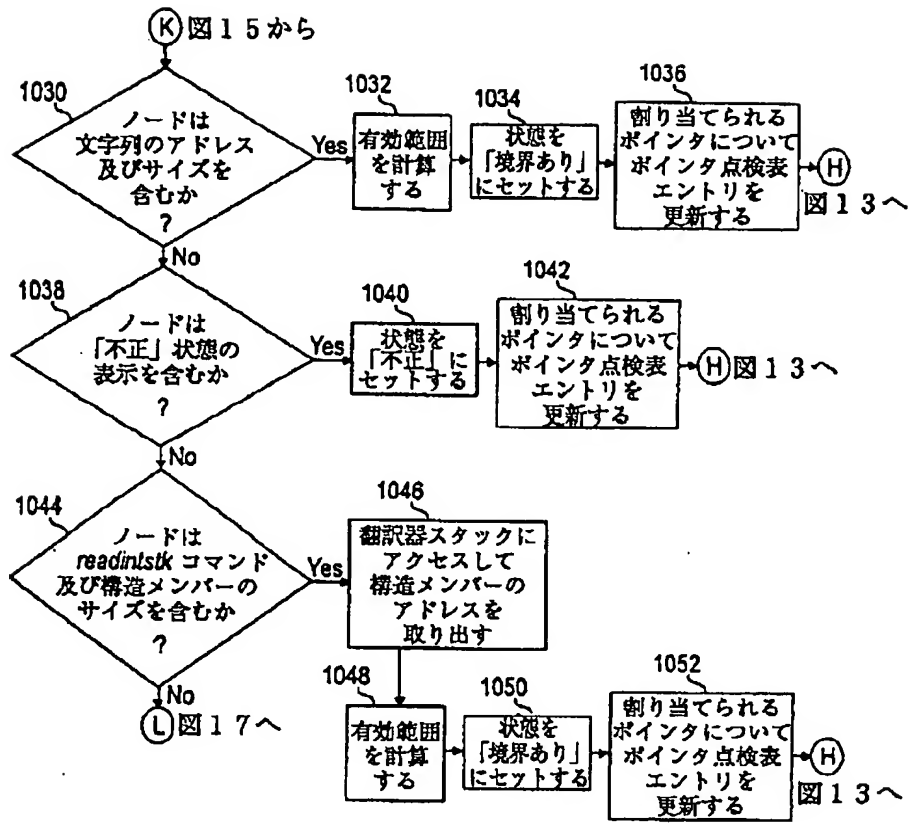
【図13】



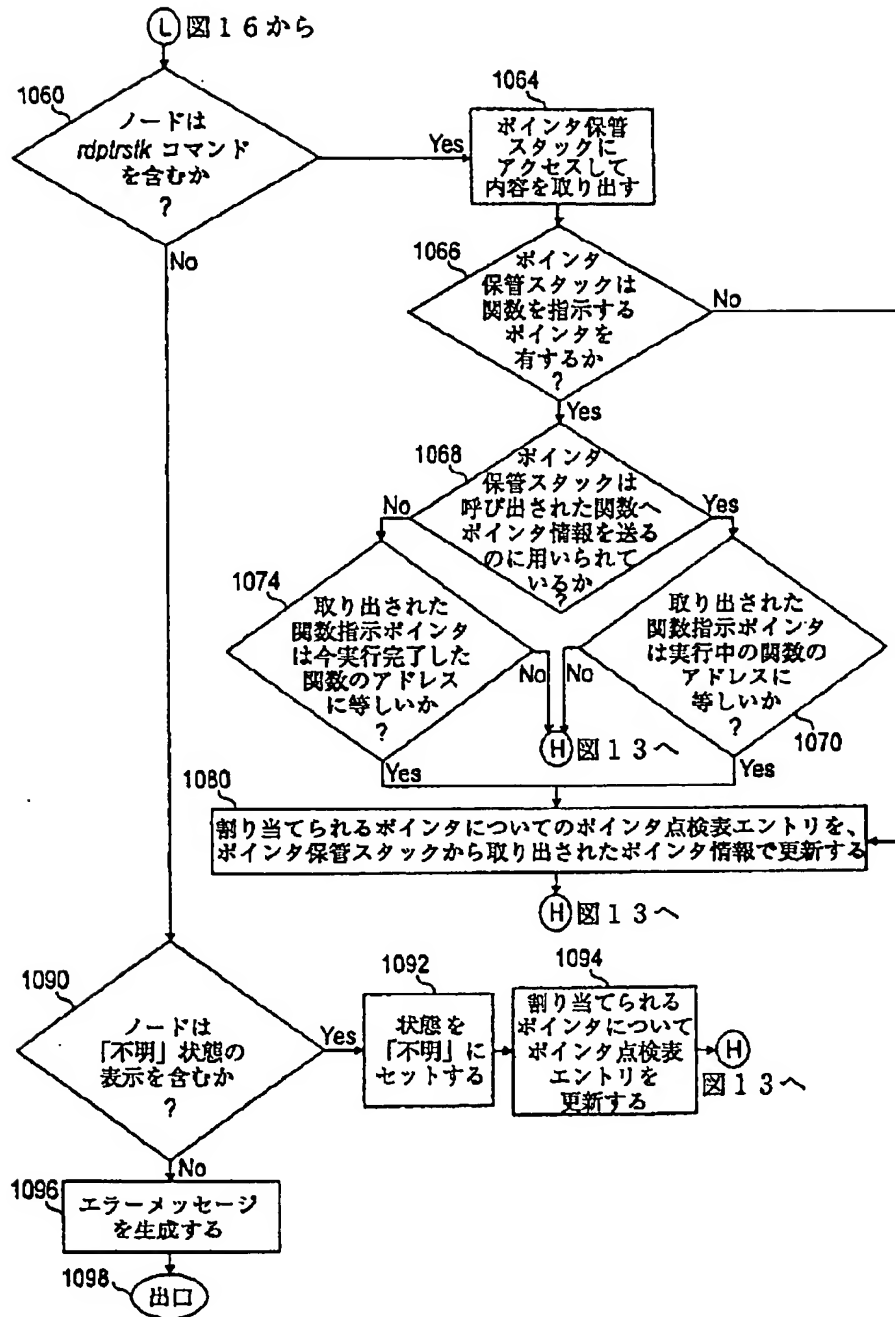
【図15】



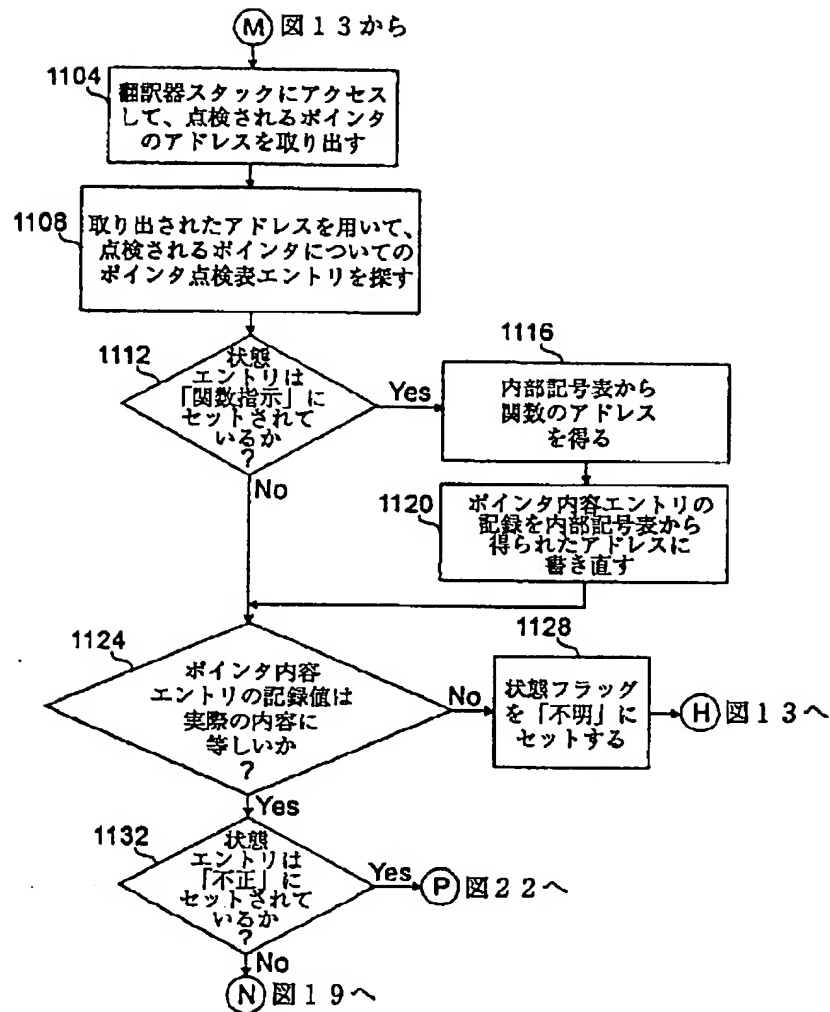
【図16】



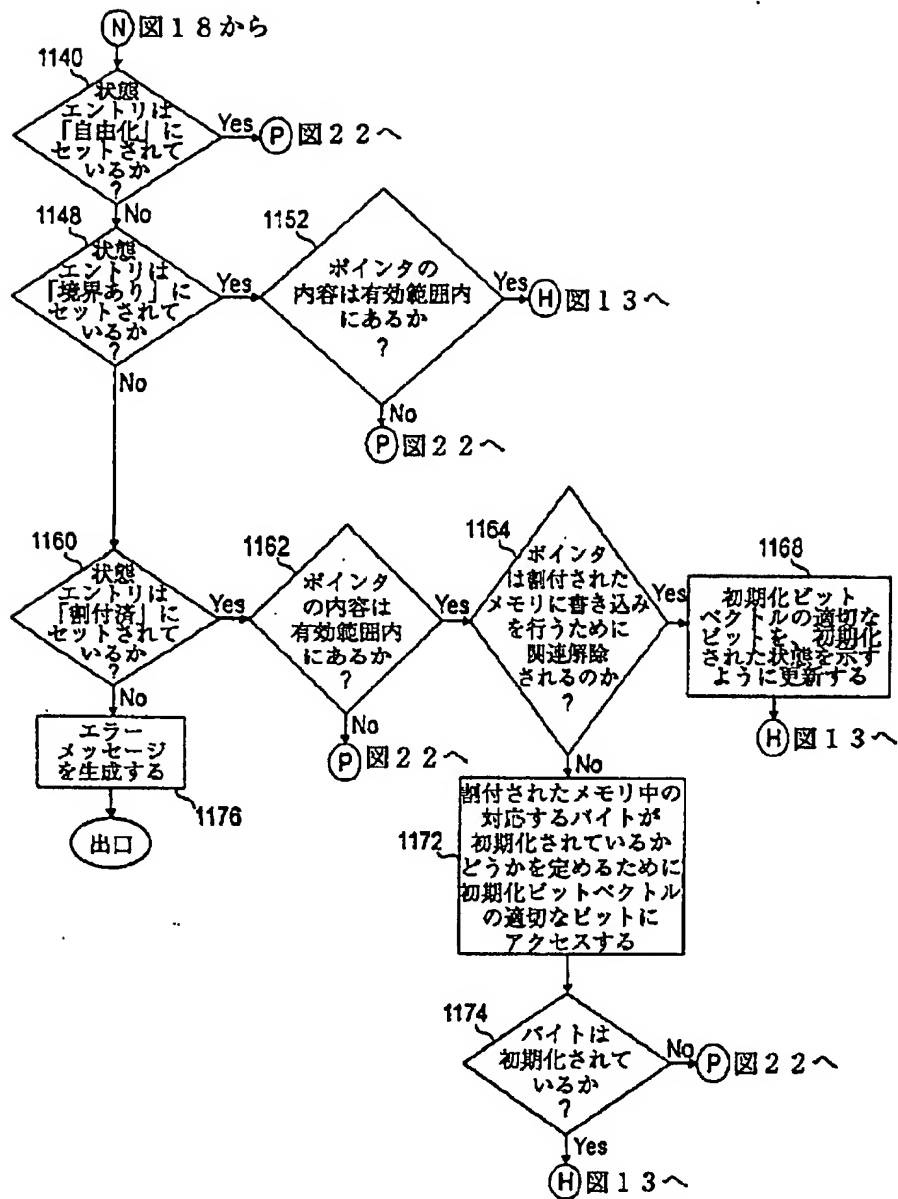
【図17】



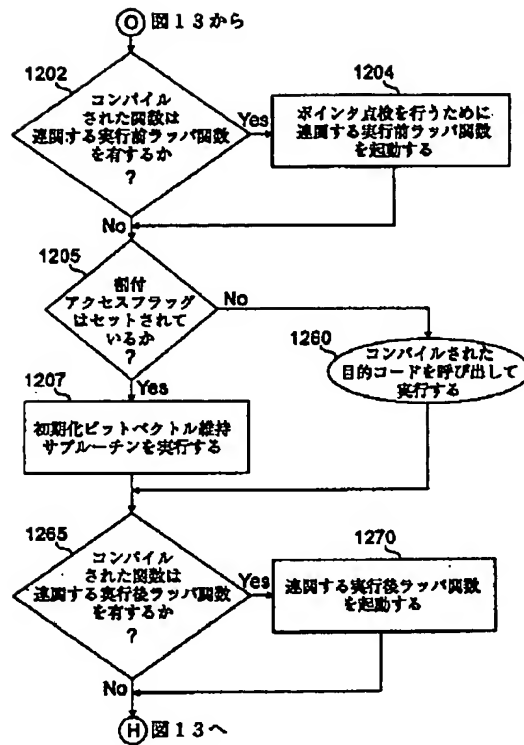
【図18】



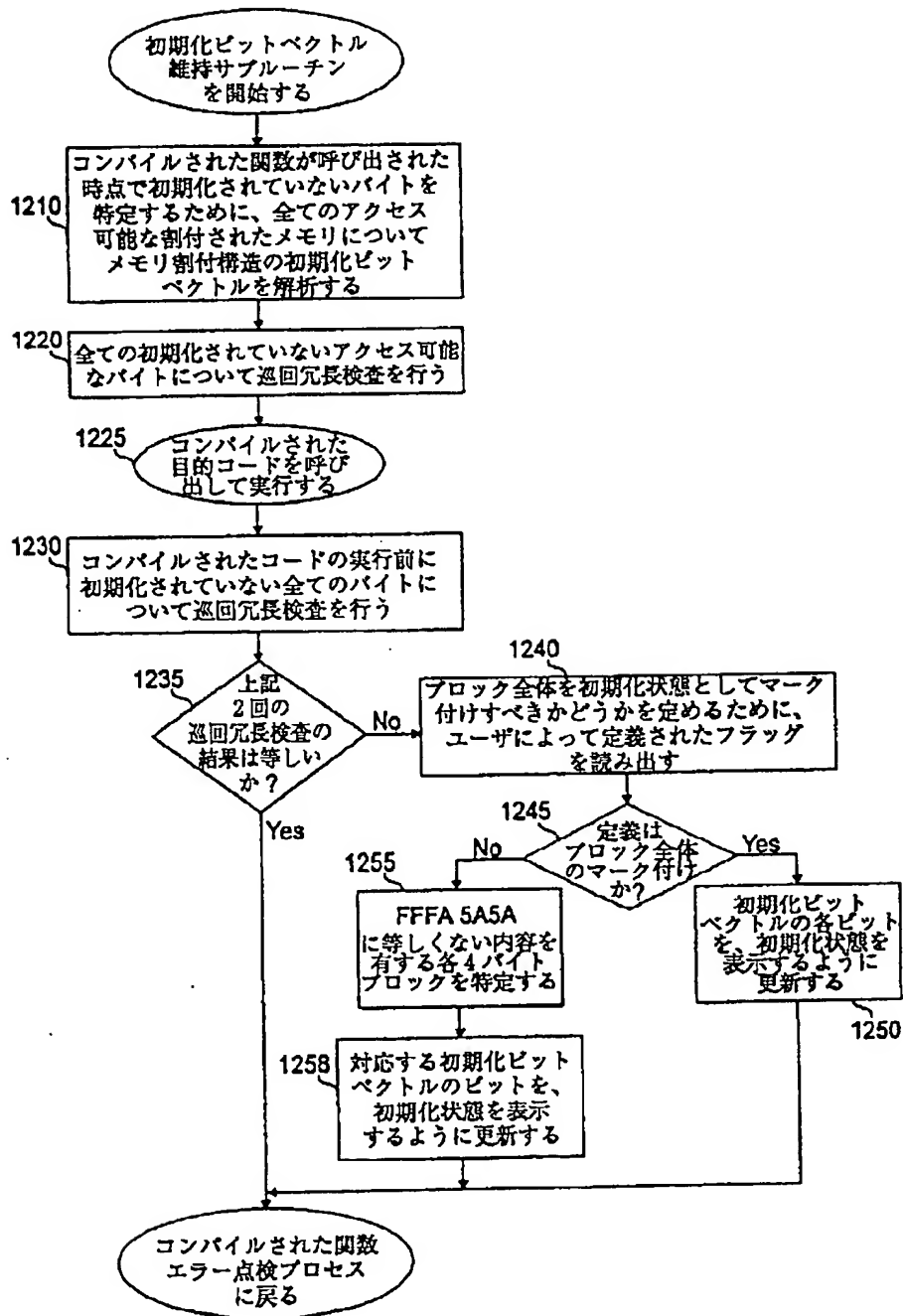
【図19】



【図20】



【図21】



フロントページの続き

(71)発明者 サディアス ジュリアス コワルスキー
 アメリカ合衆国、07901 ニュージャージー
 ー、サミット、ストーンリッジ ロード
 73

(72)発明者 ジェームズ アール、ローランド
 アメリカ合衆国、07078 ニュージャージー
 ー、ショート ヒルズ、サッカレー ドラ
 イブ 18